DESIGN OF A DISTRIBUTED MEMORY UNIT FOR CLUSTERED MICROARCHITECTURES

Stefan Bieschewski

Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya
Barcelona (Spain), April 2013

Advisors:
Antonio González
Joan-Manuel Parcerisa

A THESIS SUBMITTED IN FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
Doctor en Informàtica

ABSTRACT

Power constraints led to the end of exponential growth in single–processor performance, which characterized the semiconductor industry for many years. Single–chip multiprocessors allowed the performance growth to continue so far. Yet, Amdahl's law asserts that the overall performance of future single–chip multiprocessors will depend crucially on single–processor performance. In a multiprocessor a small growth in single–processor performance can justify the use of significant resources.

Partitioning the layout of critical components can improve the energy–efficiency and ultimately the performance of a single processor. In a clustered microarchitecture parts of these components form clusters. Instructions are processed locally in the clusters and benefit from the smaller size and complexity of the clusters components. Because the clusters together process a single instruction stream communications between clusters are necessary and introduce an additional cost.

This thesis proposes the design of a distributed memory unit and first level cache in the context of a clustered microarchitecture. While the partitioning of other parts of the microarchitecture has been well studied the distribution of the memory unit and the cache has received comparatively little attention.

The first proposal consists of a set of cache bank predictors. Eight different predictor designs are compared based on cost and accuracy. The second proposal is the distributed memory unit. The load and store queues are split into smaller queues for distributed disambiguation. The mapping of memory instructions to cache banks is delayed until addresses have been calculated. We show how disambiguation can be implemented efficiently with unordered queues. A bank predictor is used to map instructions that consume memory data near the data origin. We show that this organization significantly reduces both energy usage and latency.

The third proposal introduces Dispatch Throttling and Pre-Access Queues. These mechanisms avoid load/store queue overflows that are a result of the late allocation of entries. The fourth proposal introduces Memory Issue Queues, which add functionality to select instructions for execution and re-execution to the memory unit. The fifth proposal introduces Conservative Deadlock Aware Entry Allocation. This mechanism is a deadlock safe issue policy for the Memory Issue Queues. Deadlocks can result from certain queue allocations because entries are allocated out-of-order instead of in-order like in traditional architectures. The sixth proposal is the Early Release of Load Queue Entries. Architectures with weak memory ordering such as Alpha, PowerPC or ARMv7 can take advantage of this mechanism to release load queue entries before the commit stage. Together, these proposals allow significantly smaller and more energy efficient load queues without the need of energy hungry recovery mechanisms and without performance penalties.

Finally, we present a detailed study that compares the proposed distributed memory unit to a centralized memory unit and confirms its advantages of reduced energy usage and of improved performance.

To my wife and parents.

ACKNOWLEDGMENTS

First, I would like to thank my advisors Antonio González and Joan Manuel Parcerisa for their support and their patience. Without them, this work would not exist.

I want to thank all the PhD students room C6-E208, which became a second home to many of us.

I want to thank my parents for their support and their faith in me. Their support allowed me to finnish this work. I want to thank my wife for her love and understanding during these years.

This work was supported by Intel Corporation and by the Spanish Ministry of Science and Education under grants TIN 2004-03072 and TIN 2004-07739-C02-01, by the Ministry of Education and Science under grant TIN 2007-61763, by the Ministry of Economy and Competitiveness under grant TIN 2010-18368, and by the Generalitat of Catalonia under grants 2005-SGR-00950 and 2009-SGR-1250.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1 Background and Motivation

1.1.1 Power Consumption

The last decade saw the end of exponential growth in single–processor performance. This end was caused mainly by power constraints. Since the adoption of CMOS technology, classic CMOS scaling [Den74] allowed engineers to shrink physical structures and to increase clock frequencies and at the same time to hold power density constant. However, classic CMOS scaling focuses on the MOS transistor and does not account for a number of additional factors.

Interconnects and Clock Frequency

New circuits and more aggressive microarchitectures allowed engineers to increase clock frequencies even faster than the sole transistor speed would have allowed otherwise. More complex microarchitectures required more and faster interconnects. Additional levels of metal where incorporated to provide these interconnects,, which increased capacitance. To speed up interconnects, they were made taller, again increasing capacitance. Tall and dense interconnects lead to increased side–to–side capacitance and thus further contributed to the phenomenon. Together, increased clock frequency and increased capacitance both led to an

unforeseen rise in power density. At some point, power density rose so much that cooling became infeasible. [Ful11]

Voltage Scaling

Classic CMOS scaling anticipates that voltages scale linearly with the scaling factor. [Den74] However, when the threshold voltage shrinks linearly, the leakage current grows exponentially. While this has been no problem 15 years ago, it became an important concern during the last decade as leakage became a significant part of overall power. To limit leakage, voltages have been reduced slower as predicted by the scaling model. Higher–than–predicted voltages lead to slower switches, which further contributes to the gap between predicted and attainable performance. [Ful11]

1.1.2 Chip Level Multiprocessing

To be able to continue to deliver an exponential performance increase in the face of recent power restraints, the semiconductor industry has adopted chip level multiprocessing. Using multiple less aggressive processor cores, it is possible to continue to increase overall chip performance. This is feasible even in the presence of severe power constraints. However, because chip level multiprocessing is not transparent to software, applications have to be rewritten to make use of multiple processors. Applications that are not rewritten will only see a modest increase in performance, unlike the exponential performance growth of the past, which required (almost) no help from software.

Amdahl's Law

A serious challenge that all parallel applications have to face is Amdahl's law. It states that even a relatively small sequential portion of an application will seriously limit the speedup that is attainable with parallelization. [Amd67] Therefore, to attain the highest overall chip performance it is of utter importance not to neglect single thread performance. In this context, even tiny improvements to single thread performance can justify the use of considerable resources. [Hil08][Woo08]

The existence of many sequential legacy applications and the ramifications of Amdahl's law for parallelized applications both stress the importance of single thread performance for the overall chip performance of chip level multiprocessor.

1.1.3 Complexity and Wire Delay

To improve the single thread performance underlying parallelism like instruction level parallelism has to be uncovered. Extracting this parallelism is increasingly difficult.

Complexity[1] is affected to a great extent by the *size* and the *width* of structures like issue queues and the corresponding forwarding logic. While larger and wider structures can exploit more instruction level parallelism, the increased complexity will ultimately limit performance. [Pal97]

Wire delay poses another challenge for microarchitectures. Wire delay does not scale with transistor delay and instead shrinks at a slower rate. Since die area has remained approximately constant, global wires have to span more gates with each process generation. To mitigate this effect and speed up wires, more and bigger levels of metal were added in the past. The wires themselves became taller and more prone to crosstalk. This lead to an increment in capacitance and contributed to the increase of power density. (See Section 1.1.1)

1.1.4 Clustered Microarchitectures

Single thread performance remains an issue of uttermost importance. One way to enhance single thread performance is to improve the instruction level parallelism. To achieve this goal, key components of the superscalar microarchitecture have to grow in size and width. The physical growth of the structures implies a corresponding growth in complexity. Clustered microarchitectures manage this complexity by partitioning the microarchitecture into small and simple units, called clusters.

If no attempt were made to control the growing complexity, it could affect the cycle time negatively and thereby nullify the performance gain that was obtained through the increased parallelism. Pipelining cannot hide the additional delays of larger and wider structures because of tight loops, which prevail in modern architectures. Well-studied examples are the issue queues and the operand-forwarding network. [Pal97][Sta00]

Instead of large and wide monolithic structures, clustered microarchitectures are formed out of multiple small and narrow structures. These structures are combined into clusters to allow fast, local communications. Global communications between clusters are more expensive but they are also explicit to the microarchitecture, which allows it to proactively minimize global communications.

As a consequence of the partition in many smaller structures and because of global communications, clustered microarchitectures achieve slightly less IPC than monolithic microarchitectures of the same size and width. Nevertheless, this disadvantage is more than compensated for by the lower complexity of a clustered microarchitecture, which allows either a higher clock frequency for architectures of the same size and width or alternatively larger and wider structures at the same clock frequency. [Pal97]

---

[1] Complexity can be defined in many ways. Here we adopt the definition by Palacharla, Jouppi and Smith. [Pal97]. They define complexity as the delay through the critical path of a circuit.

The inherently lower complexity of clustered microarchitectures also results in an improved energy efficiency compared to monolithic architectures, especially for large and wide configurations. [Zyu00][Zyu01] Because the performance of todays chips is limited by power consumption, improved energy efficiency directly translates into better performance.

1.2 Energy and Performance of Distributed Caches

This thesis focuses on the distribution of the memory unit and the data cache. The data cache can be organized in clusters just like other complex parts of the microarchitecture. Organizing both—backend and cache—in clusters has the potential to minimize communication and lower overall complexity. To further motivate this argument we use a simple mathematical model to compare a centralized and two distributed caches.

1.2.1 Modeling Caches and Interconnections

We estimate energy and latency of a data cache access of load instruction. Besides the actual cache access we also take the communication cost into account.

$$E_{\mathrm{Load}} = E_{\mathrm{CacheRead}} + E_{\mathrm{Communication}}$$
$$t_{\mathrm{Load}} = t_{\mathrm{CacheRead}} + t_{\mathrm{Communication}}$$

We model the caches and the interconnection network between clusters using CACTI 6.5 [Mur09] and compare a centralized cache with a size of 64Kbytes and two cache ports to a distributed cache with the same total size, which is partitioned into four banks of 16Kbytes with two ports each[2]. The caches are optimized to minimize delay and dynamic energy consumption.

The cost of communication depends linearly on the distance, which address and data have to travel. Assuming that clusters are laid out like in Figure 1.1 the total distance is always a multiple of the distance between two neighboring clusters. We use this distance as unit and calculate time and energy for it using CACTI.

$$E_{\mathrm{Communication}} = (d_{\mathrm{LoadAddress}} + d_{\mathrm{LoadData}}) \cdot E_{\mathrm{UnitDistance}} \cdot N_{\mathrm{BitsPerMessage}}$$
$$t_{\mathrm{Communication}} = (d_{\mathrm{LoadAddress}} + d_{\mathrm{LoadData}}) \cdot t_{\mathrm{UnitDistance}}$$

The distance address and data have to travel depends on the layout and on the distribution of data and instructions. We assume that address generations as well as cache accesses are distributed uniformly over all clusters. Figure 1.1 shows two example layouts of the back-end with a distributed and with a centralized data cache. The rectangles labeled L1D $ represent the first level data cache; the rectangles labeled ICN represent the interconnection

[2] These parameters give the distributed cache four times as much bandwidth. We chose the same distributed cache parameters for this experiment as we used in the experiments for Chapter 4 and 5 of this thesis. A centralized cache with equal bandwidth would result in an extremely slow and energy-hungry cache. Hence, we restrict the number of ports of the centralized cache to two.

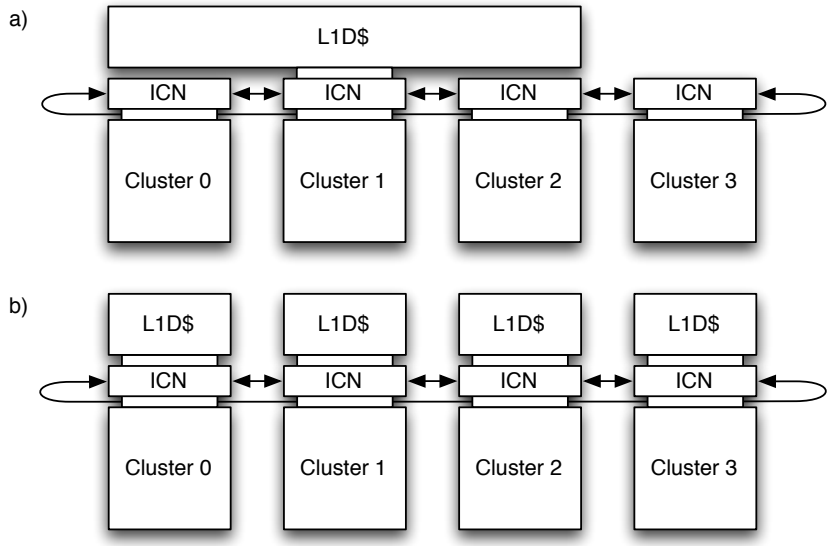

Figure 1.1: Example layouts for the back-end. Example a) shows a distributed cache where each cluster can access directly part of the cache, while example b) shows a centralized cache where only Cluster 1 can access the cache directly. ICN denotes the interconnection network, which allows clusters to access remote caches.

network. [Par04] We assume a topology of a bidirectional ring where all links have the same priority (even though the link between clusters 0 and 3 has a higher cost than the other links). There are three configurations:

- A **centralized configuration** with layout a) from Figure 1.1 above. In this configuration the address travels from the address generation unit to cluster 1, the cache performs the read access, and the load data travels back to the destination cluster, where instructions wait to consume the data. Addresses and data travel one and a quarter side lengths each on average.
- A **naïve distributed configuration** with layout b) from Figure 1.1. above. The address travels from the address generation unit to the data cache, the cache performs the read access, and the load data travels back to the destination cluster, where instructions wait to consume the data. Address and data travel one and a half side lengths each on average.
- An **ideal distributed configuration** using the same layout b) from Figure 1.1 as the naïve configuration above. This configuration always maps the consumers of load instructions to the clusters where the data reside[3]. The address travels from the address generation unit to the data cache, and the cache performs the read access. Because the data is consumed in the same cluster, no further communication is necessary[4]. Only the address travels one and a half side lengths on average, load data is always consumed locally.

[3] This is an idealization, because the address (and therefore the destination cluster) is not yet known in the pipeline stage were registers are mapped to clusters. A real architecture might utilize a predictor to provide this information in time. This configuration assumes a perfect predictor.

[4] Chapter 4 will describe the communication scheme in more detail.

$$d_{\mathrm{LoadAddress}} = \begin{cases} 1.25, & \text{if centralized} \\ 1.5, & \text{if naïve distributed} \\ 1.5, & \text{if ideal distributed} \end{cases}$$

$$d_{\mathrm{LoadData}} = \begin{cases} 1.25, & \text{if centralized} \\ 1.5, & \text{if naïve distributed} \\ 0, & \text{if ideal distributed} \end{cases}$$

To calculate the length of the interconnects between two neighboring clusters we roughly estimate the chip area of each cluster. We studied illustrated die plots of three out-of-order microarchitectures (MIPS R10K, Intel Pentium III, and IBM PowerPC 7400) and measured the chip area of the blocks that correspond roughly to a cluster of the back-end. These blocks account for two integer units, a single floating-point unit, the memory unit, the register file and the corresponding issue queues. By averaging the three estimates, we arrive at an approximate area of 1600 million square lambda[5] per cluster or at a square with a side length of 40 thousand lambdas.

$$N_{\mathrm{BitsPerMessage}} = 78$$

Finally, interconnects are 78 bits wide, 64 bits of address or data and 14 bits of additional information like instruction type, destination cluster, sequence number, access size, etc.

1.2.2 Energy and Performance Estimations

Figure 1.2 shows the normalized average access time per read access and the normalized dynamic energy usage for all three configurations and for four feature sizes.

The left graph of Figure 1.2 shows a significantly lower average access time for the two distributed configurations compared to the centralized configuration. The difference between distributed and centralized configurations increases with newer process technologies. The average access times are dominated by the cache access and the smaller distributed caches have the shorter access times. The right graph of Figure 1.2 shows the dynamic energy usage. The ideal distributed configuration uses significantly less dynamic energy than the other two configurations. The dynamic energy usage is dominated by the interconnects and the ideal distributed configuration only sends load addresses but no load data over the network. The static energy consumption (not shown in Figure 1.2) is slightly less (7 to 12%) for the distributed configurations.

We deduce from these estimations, that a distributed cache organization has the potential for shorter access times and for a significantly lower dynamic energy usage. The advantage exists for all process technologies we examined and is more pronounced in newer technologies.

---

[5] Lambda is the unit defined as half the process size.

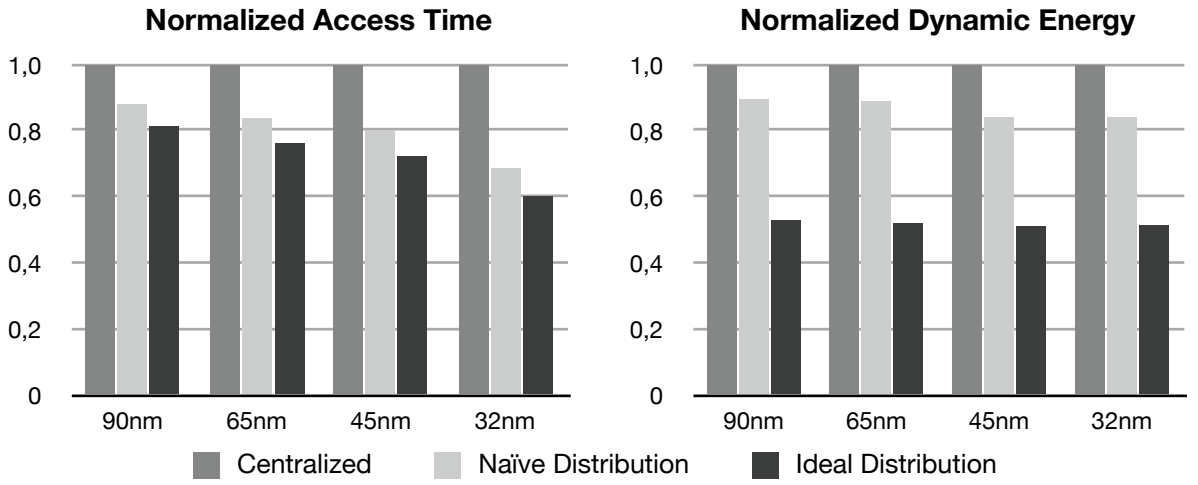

Figure 1.2: Normalized Access Time and Dynamic Energy. These diagrams show the average normalized access times and the average normalized dynamic energy for the configurations described in detail in Section 1.2.1. The data include the cache access as well as the intercommunications between clusters. The x-axis shows data for four different process technologies.

Furthermore, comparing the naïve and the ideal distributed configurations, we can conclude that it is worthwhile to minimize communications between clusters. A bank predictor (see Section 1.3 below) can help to reduce these communications. A configuration with a good bank predictor will approach the communication characteristics of the ideal configuration and share its benefits. (See Section 3.6 for concrete numbers.)

This analysis focused on the distribution of the data cache. However, the entire memory unit will benefit from a distributed implementation and the corresponding reduction in complexity. This is especially true for the disambiguation logic, which contains large content addressable memory structures. A distributed implementation can use smaller and narrower structures, thereby significantly reduce the complexity and achieve shorter access times as well as lower dynamic energy consumption.

1.3 Thesis Overview and Contributions

This section will give an overview over this thesis.

1.3.1 Bank Predictors

Chapter 3 explores bank predictors. As we have seen in Section 1.2 bank predictors are very useful to reduce global communications in clustered microarchitectures with a distributed data cache. Using the bank predictions, the instruction steering mechanism can map instructions to the cluster where their memory data resides. Since the actual memory address of a load instruction is calculated late in the pipeline, the bank is not yet known when the load instruction and its consumers pass the steering stage. The bank predictor makes this information available on time.

We adapt several well-known branch and value predictors to the domain of bank prediction and study their accuracy as well as their complexity. Previous work on bank predictors was either limited to two banks [Yoa99] (we study predictors with eight banks) or centered on the general feasibility of the approach without considering complexity. [Nee00] We find that some hashing schemes adopted from value predictors by other researchers [Bal02] are not effective.

The best predictors we found in our study were the Gshare and Tournament Predictors, based on the well-known branch predictors [McF93]. An accuracy of 60% can be obtained with a Gshare predictor of only 192 bytes size, an accuracy of 82% requires a Gshare predictor of 3Kbytes, an accuracy of 90% requires a Tournament Predictor of 15Kbytes size and finally an accuracy of 95% would require a huge Tournament Predictor of 652Kbytes.

None of the many predictors we tested could deliver a satisfactory confidence prediction for the bank predictions. Other proposals for distributed memory units [Zyu00][Zyu01] [Yoa99] use a confidence predictor (in addition to the bank predictor) to control speculation and reduce the number of mispredictions. If their microarchitectures have no confidence in a bank prediction, they do not use the bank prediction speculatively and thereby avoid the risk of a misprediction. In this case, they send the instruction to all memory pipelines instead of just the pipeline indicated by the bank predictor. Mispredictions can still occur when the microarchitecture has confidence in an incorrect bank prediction. The authors of these proposals assume that less than 3% of all confident predictions are mispredicted and therefore misspeculations do not impact performance much, even if the cost of a single misspeculation is significant. In contrast, because we are unable to reproduce a comparable confidence predictor for our distributed memory unit we forego the use of a confidence predictor altogether and strive instead to minimize the cost of each single bank misprediction.

1.3.2 A Distributed Memory Unit

Chapter 4 describes our proposal for a distributed memory unit. We propose the design of a distributed memory unit, which contains some novel contributions. We propose a steering scheme and an associated memory pipeline that make use of a bank predictor but do not rely on the accuracy of a confidence predictor and instead tolerate mispredictions gracefully.

We propose physically unordered load and store queues as well as the late allocation of queue entries. (See Bieschewski et al. [Bie07]) A similar approach was simultaneously proposed by Sethumadhavan et al. [Set07] and we adopt their terminology to avoid confusion. In our case, it is especially attractive to delay the allocation of queue entries until the address calculation, because it allows us to tolerate bank mispredictions gracefully. Unordered queues are a direct consequence of late allocation, the only two alternatives would be very sparsely occupied queues or a complex entry compression scheme, both alternatives are very expensive. We show how unordered queues can be adopted using several previously known techniques.

We propose schemes to allocate and free entries in the unordered queues of the distributed memory unit as well as schemes to handle store address and store data instructions[6].

1.3.3 Improvements to the Distributed Memory Unit

We introduce techniques to improve the flow of memory instructions. These techniques can stall the dispatch stage to control the flow of memory instructions. This reduces the pressure on some structures (like buffers of the interconnect network) that can lead to overflows. It also reduces the number of deadlock events, which can occur because queue entries are allocated out-of-order.

To improve performance and to allow more flexible memory issue policies we introduce memory issue queues to our design. These queues allow the choice of an issue policy. This can be used to give older instructions precedence over younger instructions, which improves performance. Memory issue queues also allow us to propose a conservative issue policy, which completely eliminates deadlock events. This issue policy collects information about local and remote instructions to decide when a memory instruction can safely issue (and allocate an entry in the load or store queue) without causing a deadlock.

The last proposal allows load queue entries to be safely released before the commit stage. This technique increases the effective window size for memory instructions of the processor.

1.3.4 Thesis Contributions

- Bank Predictors

We propose various bank predictors. The most promising predictors are based on branch predictors and are adapted to deliver 3 bit bank predictions instead of 1 bit branch predictions. The resulting bank predictors outperform bank predictors of comparable size based on value or address predictors.

- Distributed Memory Unit

We propose the late allocation of queue entries and physically unordered queues for a distributed memory unit. We also demonstrate how these queues may be implemented with a complexity that is comparable to conventional queues.

We propose a steering scheme and a interconnection network for memory addresses and data. This scheme minimizes communication between clusters using a bank predictor without relying on a confidence prediction.

---

[6] Our microarchitecture proposal splits store instructions in two internal instructions: a store address and a store data instruction, like e.g. the Intel P6, see Section 2.2.3.

- Dispatch Throttling

  We propose a scheme to control the instruction flow at the dispatch stage. This scheme reduces the pressure on the queues of the distributed memory unit and thereby reduces deadlocks and overflows.

- Memory Issue Queue

  We propose a scheme to issue memory instructions using a memory issue queue. The memory issue queue allows older instructions to issue before younger instructions. This priority scheme improves performance and it allows the memory issue queue to stall instructions without generating a deadlock inside the issue queue. This property of the memory issue queue is used in the next proposal.

- Conservative Deadlock Aware Entry Allocation

  We propose a scheme to intelligently issue instructions from the memory issue queue only if there is no danger of a deadlock in the memory queue. The scheme considers the status of instructions in all clusters.

- Early Release of Load Queue Entries

  We propose a scheme to release load queue entries before the commit stage. This optimization is only possible if weak memory ordering is used. It improves performance by effectively enlarging the memory queue.

- Energy and Performance Comparison

  We model energy and performance of a centralized and a distributed memory unit. The model confirms that our proposal uses significantly less energy and exhibits a higher performance.

1.4 Document Organization

- Chapter 2 gives a detailed overview over related research and the state of the art. Several examples are discussed in detail.
- Chapter 3 treats bank predictors. We present various predictors, explain the differences between predictors of different domains, and finally evaluate the predictors and discuss the results.
- Chapter 4 presents the distributed memory unit. The various techniques and mechanisms of the memory unit are discussed in detail.
- Chapter 5 presents the improvements to the distributed memory unit. This chapter contains four proposals on how to further improve the distributed memory unit that is presented in Chapter 4.
- Chapter 6 gives a short summary and an outlook of possible future research directions.

CHAPTER 2

PREVIOUS RELATED WORK

This chapter will give a short overview of disambiguation techniques and previous approaches to solve this problem. After a short introduction to the topic we will discuss some commercial solutions in detail to point out various challenges of an implementation. We continue with the discussion of academic proposals for distributed disambiguation and conclude establishing the relevance of bank prediction for distributed disambiguation.

2.1 Memory Disambiguation

Memory disambiguation [Fis83] is a set of techniques to identify dependencies between memory operations. This allows a processor to perform these operations in parallel or in a different order while still producing the expected results. This flexibility is essential to achieve high instruction level parallelism in modern processors.

The first implementations of memory disambiguation were performed exclusively in software at compile-time. [Fis83] This type of disambiguation is called static. Memory disambiguation can also be implemented at runtime [Hua94], which is called dynamic. While it is possible to perform dynamic disambiguation entirely in software it is much more suited to a combined software/hardware implementation or to an implementation in hardware only.

Static and dynamic disambiguation each have their applications. Static disambiguation allows more complex optimizations because it is performed at compile-time. Dynamic disambiguation allows optimizations based on run-time values–namely the address of memory accesses–and can change according to program behavior.

Transparent dynamic hardware disambiguation has the additional benefit that it automatically works with legacy binaries. Static disambiguation usually requires a recompilation of an application.

In the following discussion, we will restrict ourselves to discuss transparent dynamic memory disambiguation performed in hardware.

Examples of Dynamic Memory Disambiguation

Load instructions are usually more urgent than store instructions because the following instructions are more likely to depend on the outcome of a register than on the outcome of a memory location. Register dependencies are much more common than memory dependencies, loads are therefore more likely than stores to form part of the critical path.

For this reason, if a load and a store instruction compete for a resource like a cache port, it is usually beneficial to give preference to the load, even if it follows the store in the original instruction stream. However, changing the order of execution may result in unwanted results. Example 2.1 shows a code sequence consisting of a store and a load instruction. On the left hand, the addresses A and B refer to different memory locations and consequently the order of execution does not change the results. The right hand shows a different situation. Here, A and B refer to the same memory location. If the load is executed before the store, the resulting value in the register r2 is incorrect. The function of memory disambiguation is to detect memory dependencies like those on the right side to avoid the incorrect result shown in this example.

		A and B refer to different memory locations	A and B refer to the same memory location
Initialization:		*A=a,*B=b, R1=1, R2=2;	A=B,*A=*B=ab, R1=1, R2=2;
Original Program:		Store R1, A Load R2, B	
Final State	Original Order:	*A==1,*B==b, R1==1, R2==b	*A==*B==1, R1==1, **R2==1**
	Reversed Order:	*A==1,*B==b, R1==1, R2==b	*A==*B==1, R1==1, **R2==ab**

Example 2.1: Store-Load Dependency. The expressions use a syntax resembling the C language. The symbol * denotes a reference, the symbol = an assignment, and the symbol == asserts equality. The final state depends on the execution order if the memory references conflict.

Speculative Store Instructions Require Disambiguation

Speculative processors are a common case where the order of load and store instructions is often changed. These processors execute instructions speculatively in advance and later commit changes to the software–visible processor state, when all previous instructions have been executed correctly. Load instructions can be treated like other instructions that write to registers but store instructions that write to main memory must not execute speculatively. If a store would overwrite main memory, it could not easily be undone in case of a misspeculation. Therefore, store instructions are held in a special buffer until they commit. Loads that are executed speculatively must be disambiguated with the stores in this buffer, otherwise, incorrect results like those demonstrated on the right side of Example 2.1 can occur.

2.1.1 Disambiguating Load Instructions

In general, the disambiguation of a load instruction is necessary whenever a store instruction is delayed with respect to the load. This not only occurs in out-of-order processors, but also might happen whenever store instructions are held in a buffer. The most typical example is the store queue of a modern speculative out-of-order processor. Other common examples of these buffers include write–combine buffers and buffers in the memory hierarchy in general. To be able to correctly detect such dependencies the disambiguation logic must be able to determine the original program order of all memory instructions.

If the disambiguation logic detects a dependency between a load and such a delayed store, there exist several options to handle the conflict and avoid incorrect results. The first option is to *delay the load* as well. This implies some form of buffer where the load instruction can be held until it can advance. In case of a modern speculative out-of-order processor, this buffer is the load queue or the issue queue. If no buffer is available to hold an offending load instruction, the processor has to avoid an incorrect result in other ways, e.g. by restarting the whole pipeline. The delaying of the load always implies the existence of some mechanism to decide when a load should be released and finally executed. This mechanism may simply check periodically if the conflict persists. A more sophisticated mechanism may release the load only if a certain event occurs and the conflict disappears.

A second option to handle dependencies is to *pass the data on* from the store instruction directly to the load instruction. This technique is also called forwarding, load bypassing, and memory renaming. It requires that the store data is already available. In an out-of-order processor, this may not always be the case. Depending on the instruction architecture in question some complex corner cases can arise and complicate the implementation. It is common to delay the load instruction instead of passing the data on if one of these cases occurs. An example of such a corner case for architectures with various memory access sizes is a situation where a single load instruction depends on various store instructions at the same

time. More corner cases will be mentioned in the description of some current microarchitectures below.

In out-of-order processors there can arise situations where the address of a previously delayed store instruction is not yet available, because the instruction has not yet been executed. In such a situation the disambiguation logic cannot decide if a dependency does exist or not. Again, there are two options. The safe option is to delay the load instruction until the store address becomes known. The other option is to execute the load instruction speculatively. Of course, before such a speculatively executed load instruction can commit, the disambiguation logic must verify that no conflict did exist at the time of its execution. Processors that allow load instructions to execute in such a speculative way usually use a heuristic like a predictor to decide if the speculation is worthwhile.

2.1.2 Disambiguating Store Instructions

To maintain a meaningful memory semantic stores to the same memory location must execute in the original program order. If this requirement is not obeyed, memory contents may contain unexpected results as Example 2.2 demonstrates. While on the left side of Example 2.2 there exist no dependencies and results are identical, on the right side both stores access the same memory location, and therefore lead to an incorrect result if their relative order is changed. The easiest way to deal with this problem is to avoid reordering store instructions altogether with respect to one another. In that case, no disambiguation is necessary to detect store–store dependencies. However, if stores are allowed to execute out-of-order they must be disambiguated. Modern out-of-order microarchitectures execute stores (i.e. write store data to memory) only after the commit phase, which is guaranteed to occur in original program order. Therefore, in general these microarchitectures do not need to use disambiguation to detect store–store dependencies (see Section 2.2.4 for an exception).

		A and B refer to different memory locations	A and B refer to the same memory location
Initialization:		*A=a, *B=b, R1=1, R2=2;	A=B,*A=*B=ab, R1=1, R2=2;
Original Program:		Store R1, A Store R2, B	
Final State	Original order:	*A==1,*B==2, R1==1, R2==2	***A==*B==2**, R1==1, R2==2
	Reversed order:	*A==1,*B==2, R1==1, R2==2	***A==*B==1**, R1==1, R2==2

Example 2.2: Store-Store Dependency. The expressions use a syntax resembling the C language. The symbol * denotes a reference, the symbol = an assignment, and the symbol == asserts equality. The final state depends on the execution order if the memory references conflict.

		A and B refer to different memory locations	A and B refer to the same memory location
Initialization:		*A=a,*B=b, R1=1, R2=2;	A=B,*A=*B=ab, R1=1, R2=2;
Original Program:		Load R1, A Store R2, B	
Final State	Original order:	*A==a,*B==2, R1==a, R2==2	*A==*B==2, **R1==ab**, R2==2
	Reversed order:	*A==a,*B==2, R1==a, R2==2	*A==*B==2, **R1==2**, R2==2

Example 2.3: Load-Store Dependency. The expressions use a syntax resembling the C language. The symbol * denotes a reference, the symbol = an assignment, and the symbol == asserts equality. The final state depends on the execution order if the memory references conflict.

In the same way in which delayed stores can violate dependencies, delayed loads can lead to incorrect results. Example 2.3 shows the result of a delayed load. On the left side no dependence exists and results are unaffected by the order in which instructions are executed. On the right side a dependency exists and reordering instructions leads to an incorrect result of the load instruction. In a modern out-of-order microarchitecture this case does not occur because loads are either executed speculative or at commit while stores are always executed at commit. This mechanism prevents loads from being delayed relative to stores. If memory operations are reordered at another point of the memory pipeline however, care must be taken not to allow stores to pass loads without proper disambiguation.

2.1.3 Multiprocessor Memory Consistency

Dynamic hardware disambiguation allows a processor to exploit instruction level parallelism, while it maintains the programing model of a simple sequential processor. This model of a sequential processor is also the basis of memory consistency models of shared memory multiprocessors. To maintain the appearance of sequential processors in the presence of shared memory, the disambiguation logic must track additional types of dependencies. While the details depend on the memory consistency model of the architecture in question, many implementations track dependencies between load instructions as well as dependencies between store instructions in addition to the dependencies illustrated above.

2.2 Monolithic Disambiguation

2.2.1 First Approaches to Dynamic Hardware Disambiguation

The IBM System/360 Model 91 [And67][Bol67] from 1967 is best known for Tomasulo's Algorithm but it also features a store queue that decouples the processor core from memory. This queue contains disambiguation logic and can detect store–load dependencies. If it detects such a dependency, it can forward the store data to the load instruction.

A further improvement was the proposal of the dependency matrix for HPS [Pat85]. Using a vector of the dependency matrix the disambiguation logic can quickly detect if an instruction is preceded by a store with an unresolved address. These unresolved stores only block load instructions that follow them in the original program order.

Another important proposal was the Address Resolution Buffer (ARB) in 1996 by Franklin and Sohi [Fra96]. In addition to the features of the IBM System/360 Model 91 and the HPS dependency matrix, it can execute loads speculatively in the presence of unresolved stores. The ARB can also detect if the speculation was successful or not.

Even though the ARB is more complex than other proposals, it avoids large monolithic structures. The absence of a big content addressable memory is especially worth mentioning. The ARB achieves this decentralization by a banked organization. We will discuss banked organizations and related approaches in more detail later in this chapter.

2.2.2 Commercial Implementations for Monolithic Disambiguation

In this section, we will discuss four commercial solutions to dynamic memory disambiguation. The following information comes from public company sources. However, it was not always possible to determine if a published technique was indeed implemented in the microprocessor in question. We feel that this inconvenience is of no importance for the discussion of the state of art.

The four microprocessors and especially their memory units that we will compare in this section share similar characteristics. They are all general-purpose out-of-order processors, which are prepared to be used in multiprocessor systems. However, they all solve the resulting problems in different ways.

The central problem is to make memory accesses fast. The memory unit and the first level cache contain the critical path in many microarchitectures and therefore define the time of a clock cycle. An inefficient implementation of this path would lead to a slower clock cycle and thereby slow down the entire processor. Pipelining memory accesses helps to a certain degree but a deep pipeline where every load operation has a latency of many cycles is also undesirable. To avoid this slowdown memory disambiguation happens in parallel with the first level cache access and the virtual to physical address translation. This introduces some problems because the (physically tagged) caches as well as the memory disambiguation need the translated physical address of each memory operation.

Another challenge lies in the complexity of modern memory hierarchies. To achieve high average performance, implementations are optimized for the common case but also need to handle the complex cases correctly. Examples are cache and TLB misses and faults as well as memory dependencies. If such a complex case is detected the access may have to be delayed (or interrupted and re-executed) to receive special treatment. The microarchitecture has to

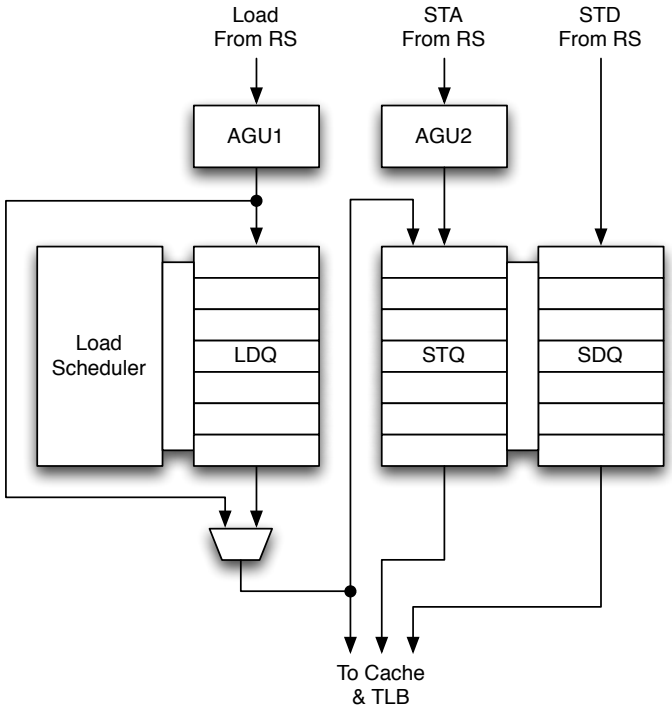

Figure 2.1: Simplified Schema of the P6 Memory Unit.

provide the mechanisms to delay (or interrupt and re-execute) these accesses. Some microarchitectures chose to speculatively execute instructions that depend on a memory operation before it can be determined if that memory operation did execute correctly. These microarchitectures need additional mechanisms to recover from misspeculations.

2.2.3 Intel P6

The first three–way superscalar out-of-order processor featuring the P6 microarchitecture was introduced in 1995 [Gwe95]. The out-of-order processor core does not deal directly with the instructions of the CISC ISA but instead operates on μ-ops, which are RISC–like operations. These μ-ops are generated on the fly from the CISC instructions by the front-end of the processor. All memory accesses by CISC instructions are translated into load and store μ-ops. While there is only one load μ-op, there are two types of store μ-ops: one that computes the store address (STA), and another that forwards the store data (STD). Figure 2.1 shows the mechanism used to schedule memory instructions. All memory operations are first issued by the reservation station (RS) to the address generation units (AGU). There are two dedicated address generation units: one for load and one for store instructions.

Load instructions are inserted into the load queue (LDQ) once their effective address has been calculated by the address generation unit[7]. [Abr97] The load queue possesses its own

[7] The patent [Abr97] was filed in 1996 and lists several well known architects of the P6 as inventors. It is therefore very likely that the patent does indeed describe the P6 implementation. For the sake of this discussion it is sufficient to note that the patent describes a feasible implementation.

scheduler that selects the load instruction, which will be executed next. Loads arriving from the AGU can bypass scheduler and load queue if no other load instructions are ready to execute.

Store instructions are inserted into the store address queue and into the store data queue where they are held until the store instruction commits. Only after commit the store data is written to the data cache.

Sometimes a resource conflict or a data dependency is detected while a load is executed. In these cases the load waits in the load queue until the conflict or dependency disappears. Resource conflicts may be caused by the data cache or the translation look-aside buffer. If such a conflict is detected while a load is executing, the load execution is cancelled and the load queue entry is marked as blocked. The load queue entry also records, which resource conflict caused the block. When the resource becomes available again, all loads, which were waiting on this resource, are awakened. This mechanism is similar to a conventional issue queue in an out-of-order microarchitecture.

The load queue scheduler handles data dependencies like resource conflicts, except that queue entries include an additional reference to the dependent store instruction. This additional reference allows a more selective wakeup of loads. Several types of data dependencies are detected. A store with an unknown address prevents all younger loads from executing—the P6 does not speculate on data dependencies. A store with unknown store data blocks all younger loads with the same address. A store to some memory location blocks younger loads to the same location when the store data size is smaller than the load data size. In this case, data forwarding from the store to the load is not possible because the store does not contain all the necessary data. The load is blocked until the store writes to memory and the load finally gets its data from the cache.

Data dependencies are detected while a load is executed. To detect the dependencies the mechanism must identify relevant store queue entries that are both valid and contain stores older than the load executing. Store queue entries are allocated in a round robin scheme by instruction age to facilitate the identification of relevant queue entries. Stores that are issued consecutively are allocated adjacent queue entries, whereby in a round robin scheme the first and the last entry are considered logically adjacent. This guarantees that the relevant stores are located in adjacent queue entries. The oldest store in the queue is marked by a tail pointer[8]. The youngest store that is still older than the load instruction is identified by a coloring scheme. Upon instruction dispatch, each load is marked with the “color” (the number of its store queue entry) of the most recently dispatched store instruction. The queue entries between the tail pointer and the “color” of the load are both valid and contain the stores that are older than the load executing.

---

[8] There are two contrary naming conventions to refer to the two ends of a FIFO. To avoid confusion, we adopt the terminology of [Abr79] wherein *tail* refers to the oldest and *head* to the youngest entry.

Store–to–Load Forwarding

Sometimes it is not enough to detect the presence of data dependencies but it is necessary to select a single dependency. This is the case for store to load forwarding. When a load instruction detects one or more older store instructions with a matching virtual address, it cannot obtain its data from memory. Instead, it must either wait for the store(s) to write their data to memory or obtain its data from the store(s). The latter is preferable for performance reasons but the required logic is extremely complex if all cases are handled. The P6 handles only the most common case where load and store have identical virtual addresses and the load data size is equal to or smaller than the store data size. In this case, the load data can be obtained directly from a single store data entry. If forwarding is not possible, the load is blocked until all data dependencies disappear, i.e. until the dependent stores write to memory.

However, it is possible that more than one store queue entry matches the load address. The forwarding logic must then identify the youngest of the matches that are older than the load. The logic scans store queue entries beginning from the store queue entry indicated by the "color" of the load instruction up to the entry indicated by the tail pointer. The store queue of the P6 has only 12 entries, so that this scan can be accomplished very efficiently. The implementation might be similar to a 12–wide carry-lookahead adder circuit where the "carry" is a signal that indicates if a match occurred.

Paged Memory

The presence of paged memory adds another level of complexity to the already complex problem of memory disambiguation. The translation of virtual to physical addresses may introduce additional data dependencies, i.e. two different virtual pages may be mapped to the same physical page, which can lead to data dependencies between instructions even though these access different virtual addresses. The address translation also introduces a delay so that physical addresses are not immediately available to the memory unit. Therefore, memory disambiguation (as well as cache memory access) are initiated with the page offset of the virtual address, which is not translated and therefore immediately available. Comparing only part of the address can guarantee the absence of dependencies but can also generate false hits. For that reason the entire virtual address is compared before data is forwarded from a store to a load. False hits have no adverse effects except that they delay the execution of the affected load instructions until the matching store commits. Using the page offset for disambiguation allows the use of a significantly smaller, faster and less power hungry CAM. On the downside, some performance is lost by load instructions, which are delayed by false hits.

Multiprocessor Memory Ordering

To achieve correct memory ordering in the presence of various processors sharing the same system bus the processor snoops the bus. All external stores are checked for dependencies with local speculative loads. When a dependency is detected, the load and all following instructions (that may have consumed an incorrect input value) are flushed from the pipeline

and then re-executed. The comparison is performed with full physical addresses but it is not time critical because matches (typically) occur infrequently and the external system bus runs at a lower clock frequency than the processor core and the first level cache. The physical addresses of speculative loads are stored in the load queue. Because all speculative load instructions must be compared, load instructions can only be removed from the load queue after they commit.

2.2.4 Digital Alpha EV6

The Digital Alpha EV6 [Gwe96][Kes98] was introduced in 1998. It was the first out-of-order microprocessor to implement the Alpha instruction set. Load and store instructions are treated in a similar manner as integer instructions and share the issue queues with integer instructions. Store instructions are not split into two microinstructions (like e.g. Intel's P6 does) but are issued once their operands (address and data) are available. After they issue, load instructions are inserted into a load queue and simultaneously access the data cache. Store instructions are inserted into a store queue after being issued but do not write to the cache until they are retired. Both load and store instructions are positioned in the queues in the original program order, even though they enter the queues at issue out-of-order. [Kes98] Figure 2.2 shows a simplified Scheme of the memory unit.

Load and store instructions are only issued once and are removed from the issue queue immediately after they issue. They are only issued if there is sufficient space in the load or store queue. No other tests for conditions that may prevent a correct execution (like resource conflicts or memory dependencies) are performed *prior* to issue. If such a condition occurs after the issue stage, the memory instruction cannot easily be re-issued because it was already removed from the issue queue and its entry may already be occupied by another instruction. In this case, the entire pipeline must be restarted. The offending memory instruction and all younger instructions are removed from the pipeline. Then instructions are fetched again from the instruction cache starting with the offending memory instruction. The register mapper stores a copy of the register map for all in-flight instructions, which enables a quick recovery of the register map after a pipeline restart. Still, a pipeline restart is expensive and the EV6 uses a predictor to avoid restarts (see below). [Kes98]

Store–to–Load Forwarding

When a load instruction issues, its address and age are compared to all entries of the store queue. A read–after–write dependency exists only if the addresses match and the matching store is older than the load. If several stores satisfy these conditions, then the youngest of them holds the data required for store–to–load forwarding. This store could be identified by comparing the instruction age of all stores that satisfy the above conditions, but this would be slow and expensive. Alternatively, the physical proximity of the queue entries could be used to deduce the relative age of instructions, like described in Section 2.2.3. However, the EV6

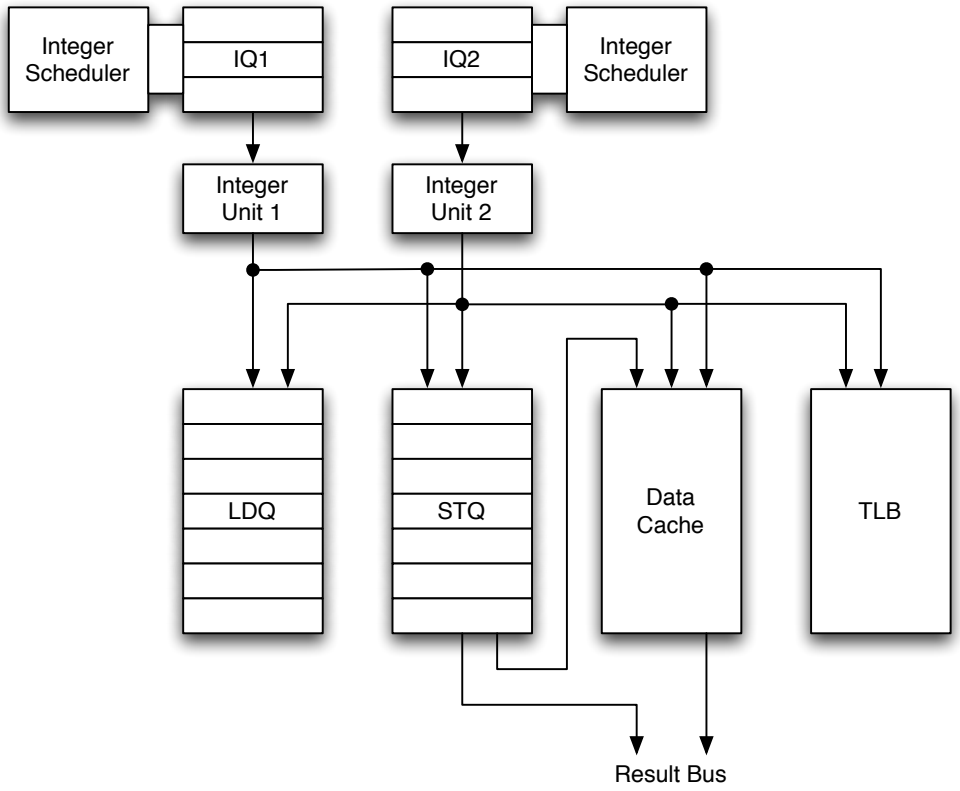

Figure 2.2: Simplified Schema of the EV6 Memory Unit.

favors another technique.[9] [Web02] Each queue entry is assigned an additional bit, called the *no–hit–bit*. When this bit is set, an entry does not report any address matches (or hits). Whenever an instruction is inserted into the store queue, the address of the new instruction is compared to all stores in the queue. If there are no matches the no–hit–bit of the new entry is cleared and the new store will participate in future searches. If there already is a matching store in the queue, its age is compared to the new store. The no–hit–bit of the older store is set and the no–hit–bit of the younger store is cleared. This assures that only the youngest store in a group of stores with equal addresses will report a match.

This solves the problem of identifying a candidate for store–to–load forwarding in the store queue when there is more than one store with a matching address. Unfortunately, there is no guarantee that the candidate store is older than the issuing load, so this condition must be checked. If the candidate store is actually younger than the issuing load, it does not qualify for forwarding. Still, there might exist dependencies on older stores in the queue with their no–hit–bit set. To resolve this situation the pipeline has to be restarted. Fortunately, this condition occurs only infrequently.

[9] The patent [Web02] was filed in 1998, lists three well known architects of the EV6 as inventors, and explicitly names the EV6 as the preferred embodiment of the invention. It is therefore very likely that the patent does indeed describe the EV6 implementation. For the sake of this discussion it is sufficient to note that the patent describes one feasible implementation.

There are more conditions like data size, alignment, etc. that have to be met for a successful store–to–load forwarding. Again, if forwarding is not possible, the pipeline has to be restarted, because there is no way to re-issue the load instruction later. These problems can be almost completely avoided by careful programming and optimizing compilers.

When a store instruction issues, its address and age are compared to all entries in the load queue. If any younger load instructions with matching address are found, the pipeline must be restarted, because these load instructions may depend on the issuing store and should have executed after the store. To avoid frequent pipeline restarts caused by ignored read–after–write dependencies the EV6 deploys a predictor, the store wait table. [Kes98] Whenever the previously described situation occurs, the predictor is trained with the Program Counter of the offending load instruction. Before load instructions are inserted into the issue queues, the predictor predicts if they will honor read–after–write dependencies. Load instructions that are predicted to violate dependencies are issued only after all older store instructions have issued.

Paged Memory

Memory disambiguation is very time critical and therefore cannot wait until addresses are translated by the translation look-aside buffer. One possible solution is to use the page offset instead of the full address, until the full physical address is available. Since the page offset is not affected by the translation, it is immediately available. Intel's P6 employs this method. The downside to this solution are false hits, which occur when two addresses seem to match but in fact only have matching page offsets. The probability of false hits increases with queue size. What's more, in case of the EV6 false hits often lead to pipeline restarts (see below) and are therefore quite expensive. To remedy this situation the EV6 uses two bits from the virtual page number in addition to the page offset for memory disambiguation. [Web02] In comparison Intel's P6 can afford to use the page offset only because it features a smaller store queue (only 12 in contrast to the 32 entries of the EV6) and the load queue scheduler reduces the cost of false hits by avoiding pipeline restarts. In addition to false hits, using the virtual address for memory disambiguation can also lead to false misses. These occur when two virtual addresses (more specifically: the subsets of the virtual addresses used for comparison) do not match but their physical addresses do match. This very infrequent case nonetheless has to be handled correctly.

To detect these situations, as soon as the physical address of a load instruction becomes available the entire store queue is searched again with the full physical address of the load. If a false hit or a false miss is detected the pipeline is restarted. Stores search the load queue to detect dependency violations that require a pipeline restart. These searches are not time critical and are performed with physical addresses.

Multiprocessor Memory Ordering

To guarantee correct memory ordering the processor has to enforce in-order execution of loads that access the same memory cell. To achieve this, all load instructions search the load queue for younger load instructions with the same address. If an offending load is found the pipeline is restarted. This search is not time critical and can be performed with physical addresses.

2.2.5 AMD K7

The first processor featuring AMD's K7 microarchitecture was the Athlon introduced in 1999. [Die98] This processor generates RISC–like operations from the complex instructions of the x86 architecture. Load and store operations are dispatched to the memory unit after they have been decoded. Certain x86 instructions where a memory operand serves as a source as well as a destination generate a single load–store operation. [AMD02]

Figure 2.3 shows a simplified scheme of the memory unit. The memory unit contains two queues to accommodate operations. Both queues are unified–they share load and store operations as well as load–store operations (load–store operations consist of a load followed by a store to the same address). The first queue (also called LS1 buffer) holds operations until their address arrives from the AGUs and they probe the first level cache. A cache probe directly returns data for load operations that hit the cache and generates a cache miss if the accessed data is not present in the cache. The first queue is rather small (12 entries) and is organized as a FIFO. Only the two oldest queue entries can probe the cache if their addresses are resolved and if cache ports are available. Once operations have probed the cache, they are moved to a second queue (also called LS2 buffer) where they reside until they commit. The second queue also serves as a scheduler for operations, which were not yet executed and were delayed (e.g. by a cache or a TLB miss).

The logic to detect memory dependencies is using a *last in buffer bit* similar to the no–hit bit of the Alpha EV6, but takes advantage of the fact that addresses stay in program order inside the two buffers. Therefore, even though the same technique is used as in the Alpha, in the Athlon it is guaranteed to always work correctly, instead of merely serving as a heuristic[10]. [Hug02]

The simple architecture and small capacity of the LS1 buffer also allow for an efficient implementation. The price for these advantages is the risk that a memory instruction with an unresolved address will hold up the whole memory pipeline.

---

[10] The patent [Hug02] was filed in 1999 and describes a microarchitecture as context, which perfectly matches AMDs official documentation of the Athlon [AMD02]. It is therefore very likely that the patent does indeed describe the Athlon. For the sake of this discussion it is sufficient to note, that the patent describes a feasible implementation.

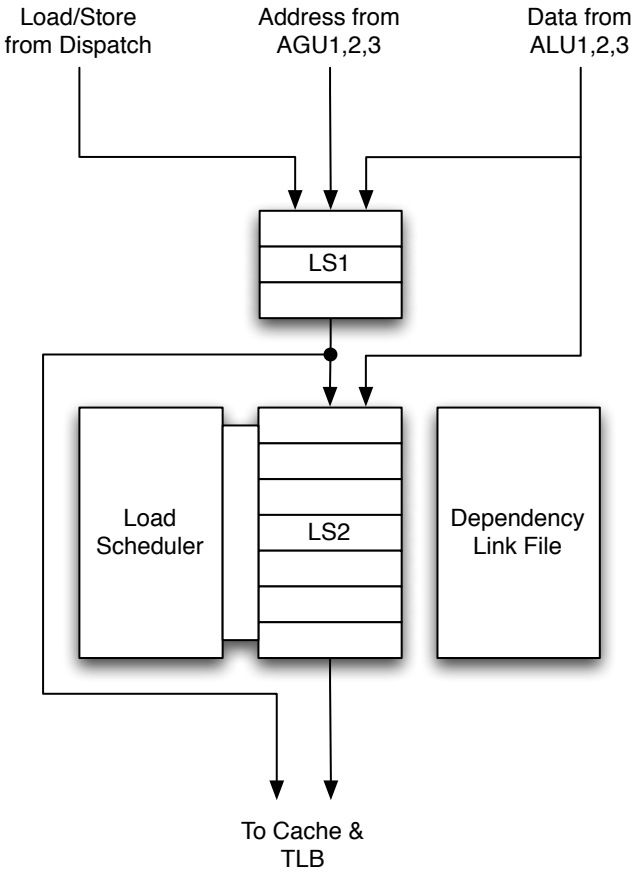

Figure 2.3: Simplified Schema of the K7 Memory Unit.

Multiprocessor Memory Ordering

Multiprocessor memory ordering for x86 compatible architecture requires that store instructions appear to occur in the same order on all processors. Stores broadcast their addresses to all processors and each address is searched in the LS2 buffer for matching load instructions. If any load instructions are found the pipeline is restarted because these loads make stores appear to execute in a different order on some processors.

2.3 Distributed Disambiguation

The division of memory in banks has been common in computer architecture for a long time. Because banks can be accessed in parallel, the maximum throughput of the entire memory increases. Ideally the bandwidth of banked memory approaches that of truly multi-ported memory even though multi-ported memory is much more expensive than memory built from single-ported memory banks. In practice however, the bandwidth is limited by bank conflicts that occur when multiple accesses to an individual single-ported memory conflict with one another.

The banked organization is useful for all layers of the memory hierarchy from disc arrays and main memory up to processor caches and registers. It is also possible to use it for internal

structures of the memory unit, like load and store queues, and substitute one big memory with several smaller memories. Because part of this memory is fully associative, its size is of special concern for complexity, speed and energy efficiency.

2.3.1 Yoaz et al.

Yoaz et al. [Yoa99] were the first to propose the use of a cache bank predictor to avoid cache bank conflicts and simplify the pipeline design. They propose to use an independent memory pipeline for each cache bank. Because memory instructions issue to the pipelines before the actual data addresses are known (the address generation units are part of the pipelines), cache bank conflicts are difficult to avoid.

Memory instructions could be scheduled a second time after their address was calculated but this is less desirable because it adds complexity to the microarchitecture and tends to increase the latency of load instructions. (Compare Figure 2.4: Conventional Multi-banked, Dual scheduled Multi-banked) Yoaz et al. propose a bank predictor to aid load instruction scheduling and to avoid bank conflicts. In addition, Yoaz et al. argue that a sufficiently accurate predictor enables simplified memory pipelines that need no crossbars or memory schedulers yet can approach bandwidth and latency of a true multi-ported cache. (Compare Figure 2.4: Sliced Multi-banked) Because the penalty of a misprediction in such a simplified pipeline is high (the load and all dependent instructions must re-execute) the accuracy of the bank predictor is crucial for this architecture. To achieve the required accuracy Yoaz et al. propose to assign a confidence to each prediction and replicate all load instructions with a low–confidence prediction to all pipelines simultaneously. If a memory pipeline is idle, loads with high–confidence predictions can be replicated too to further lower the number of mispredictions. Once the data address, and therefore the cache bank, is known, unneeded replicated instructions are cancelled. Figure 2.4 summarizes the four variants of memory pipelines discussed by Yoaz et al.

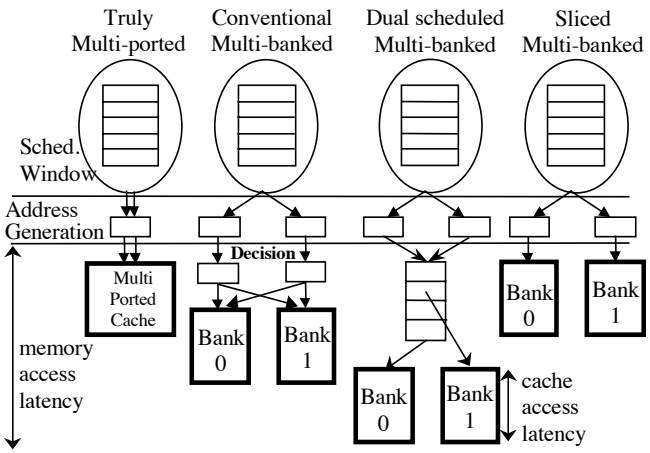

Figure 2.4: Different Memory Pipeline Organizations, Illustration from Yoaz et al.

2.3.2 Zyuban and Kogge

Just as banked organization is based on physical partition of memory structures, clustered microarchitecture (see Section 1.1.4) is based on the physical partition of microarchitectural structures. By establishing a fixed mapping between clusters and cache banks, both concepts are combined. This combination facilitates the communication between a cluster and its associated cache bank. By reducing the distance for these communications, less power is spent and the access time is improved.

Zyuban and Kogge [Zyu01][Zyu00] apply the ideas of Yoaz et al. to their proposal of a clustered microarchitecture. Each cluster of the microarchitecture is associated with a cache bank and a memory pipeline and each memory pipeline has an independent memory scheduler.

Load instructions with a high–confidence prediction are assigned to a single cluster. The load instruction calculates its effective address, and if the prediction turns out correct, it enters the memory pipeline, and accesses the cache bank, all locally in the same cluster.

Load instructions with a low–confidence prediction reserve entries in all memory pipelines at once, even though the effective address is only calculated in one cluster (Yoaz et al. also replicate address calculations in similar cases). Once the address is known, it is sent to the corresponding cluster where it enters the memory pipeline while the unneeded reservations in other clusters are cancelled.

Incorrect high–confidence predictions are a special case. Zyuban and Kogge propose to transfer the address from the cluster where it was calculated to the memory pipeline where the access has to take place and transfer the result back again to the first cluster where the result register was reserved and where the result is expected by dependent instructions. Two complications arise.

First, there might be no space for additional instructions in the destination memory pipeline. Worse, a deadlock might arise[11] and prevent any further progress of the microarchitecture. Zyuban and Kogge suggest to reserve four otherwise unused extra entries in each memory pipeline to decrease the frequency of this event and to generate a soft exception (a pipeline flush) to recover from deadlock.

Second, the disambiguation scheme requires all instructions to enter (or reserve an entry in) the memory pipelines in program order. However, instructions with an incorrect high–confidence prediction do not enter the memory pipeline in program order. Disambiguation requires not only the address but also the relative ordering of the instructions in the memory

[11] Deadlocks can occur when resources are allocated out of program order; e.g. if the oldest instruction in flight cannot progress because younger instructions hold all instances of a critical resource—in this case the entries of the load queue.

pipeline. As described above, traditionally this ordering of instructions assumes an instruction queue where instructions are stored in physically adjacent entries in program order. Inserting an instruction into this kind of structure is not trivial, but unfortunately, Zyuban and Kogge do not elaborate on this issue. The issue is nonetheless important, because the only viable alternative to handle it is an expensive pipeline flush whenever an incorrect high–confidence prediction occurs.

Zyuban and Kogge emulate[12] a bank predictor with prediction rate/accuracy percentages of 70/99 respectively. The prediction rate refers to the percentage of predictions with high confidence; the accuracy refers to the percentage of correct predictions out of predictions with high confidence. The same pair of parameters is used for the simulation of two and four cache banks citing the results of Yoaz et al. [Yoa99] However, Yoaz et al. report prediction rates/ accuracy percentages of 50/98, 50/97 and 70/97 respectively for two banks only. Zyuban and Kogge's assumptions are therefore somewhat optimistic especially when more than two cache banks are concerned.

In this thesis, we will propose alternative solutions to these problems, which do not rely on idealized components and discuss possible implementations in detail. This includes bank predictors as well as a scheme to manage the allocation of entries in distributed memory queues.

2.3.3 TRIPS

Sethumadhavan et al. [Set07] present the memory architecture of TRIPS (Tera-op, Reliable Intelligently adaptive Processing System) a project of the University of Texas at Austin. The work on TRIPS proceeded in parallel with the work of this thesis [Bie07] and independently arrived at similar conclusions on some open questions, most notably the suitability of unordered queues for distributed disambiguation and the resulting issue of flow control.

TRIPS is a clustered microarchitecture with a block–oriented instruction set where blocks of up to 128 instructions execute in an atomic manner. Up to 32 of the instructions of a block can access memory and up to eight blocks can simultaneously be in flight. TRIPS is organized in tiles of different types, which are replicated and connected by several on chip networks to form the complete microarchitecture. Among others there are *execution tiles* that contain arithmetic units (which are also used to calculate memory addresses) with the corresponding issue queues to execute instructions out-of-order and *data tiles,* which contain a data cache bank with the corresponding disambiguation logic and a joint load/store queue.

The paper stresses the fact that by delaying the reservation of entries in the load/store queue until instructions arrive at the data tile, the average occupancy of the queue can be

[12] We suppose that the term *emulate* refers to a statistical emulation of the predictor, where prediction rate and accuracy are fixed and the internal structure of the predictor is a black box.

reduced. Traditionally, entries in the memory queue are reserved in program order before addresses are calculated, to be able to store them in the same order in physically adjacent entries. By delaying the reservation, it is no longer feasible to maintain that order and the mechanisms to perform the basic functions of disambiguation must be revised. SethumadhavanM et al. propose to extend the load/store queue by an *age CAM* that can compare the age (e.g. a sequence number) of an instruction to all entries in the queue and output greater/lesser/equals results for each entry. Using this age CAM, the disambiguation logic can be implemented in a straightforward manner with one exception. If a load instruction hits several older store instructions the logic is unable to immediately order the stores by age and requires a cycle per store instruction hit to establish dependencies and forward store data to the load instruction. Traditional architectures can delay the load instruction until the older stores write to the cache and then obtain data from the cache, but TRIPS' atomic instruction blocks do not permit stores to commit and write to the data cache if they are located in the same block as a non-completed and dependent load instruction. The atomicity of instruction blocks also puts a lower limit on the size of the load/store queue of a data tile. Exactly one of instruction blocks in flight inside the processor is non-speculative, it is guaranteed to execute atomically, while the other blocks are speculative and their results might be discarded later. At any moment there can be up to 32 non-speculative and up to 224 speculative memory instructions in flight. Every data tile must be able to hold at least 32 memory instructions because all non-speculative might be directed to the same data tile (in TRIPS a data tile corresponds to data cache bank with an associated memory pipeline). A priority scheme favors non-speculative instructions over speculative instructions to guarantee forward progress of the microarchitecture. However, prioritizing non-speculative instructions is costly and involves pipeline flushes. Therefore, memory queues larger than the minimum of 32 instructions improve performance.

Sethumadhavan et al. propose three methods for control flow to avoid frequent load/store queue overflows and deadlocks. These methods do not directly compare the age of instructions to establish priorities but rather distinguish only between speculative and non-speculative instructions. Non-speculative instructions are given priority over speculative instructions to avoid deadlocks. If a non-speculative instruction arrives at a data tile with a full load/store queue the pipeline is flushed to avoid a deadlock. Speculative instructions that arrive at such a full data tile are handled by one of the following three schemes.

The first scheme re-executes speculative instructions that arrive at a data tile with a full queue. These instructions are then held at the execution tile until the oldest instruction block commits.

The second scheme buffers speculative instructions that arrive at a data tile with a full queue. The data tile is extended with a special buffer where speculative instructions wait until the next instruction block commits.

The third scheme establishes two virtual channels of different priority between execution tiles and data tiles. The channel with a higher priority is reserved for non-speculative instructions; the other channel is used by speculative instructions. If a speculative instruction encounters a data tile with a full load/store queue, it asserts backward pressure and prevents the advance of other speculative instructions from the execution tiles.

TRIPS is an aggressively parallel architecture that allows up to 1024 in-flight instructions. Consequently, the load/store queues are large–between 32 and 64 entries per cluster. The larger the queues, the less frequent are queue overflows and deadlocks. But load/store queue entries are expensive. They consume considerable energy and large queues can slow the cycle time down. The age CAM further increases the cost of each queue entry. In this thesis we will propose mechanisms, which completely avoid overflows and deadlocks with small queues and without using the costly age CAM. This makes a distributed memory unit attractive for less aggressively parallel architectures, which focus on power efficiency.

2.3.4 Others

Torres et al. [Tor05] use a banked store queue inside a monolithic microarchitecture to speculatively forward memory values. The allocation of queue entries of their distributed store queue is delayed until store instructions execute. Allocation therefore happens out-of-order. Store queue entries may be overwritten before the corresponding store commits if newer store instructions arrive and the store queue is full. Furthermore, their distributed store queue does not handle multiple matches. To guarantee correct memory semantics they use a second centralized store queue, which verifies all accesses. This strategy allows fast speculative disambiguation. The downside are occasional misspeculations and the verification/re-execution of all load instructions in the redundant centralized store queue, which incur a significant cost in area and energy usage. Our proposal in contrast limits speculative activity and does not require the verification of all load instructions.

Racunas and Patt [Rac03] propose a distributed cache for clustered microarchitectures. Instead of cache banks, they associate a set of exclusive cache partitions to the clusters, so that at any time a specific cache line can be present in only one of the cache partitions. Several local and global tables map data addresses and static instructions to cache partitions. Load and store instructions that hit the local cache of their assigned cluster can be disambiguated locally. A global store queue ensures correctness in the case of mispredicted load instructions. Their proposal allows fast disambiguation but also suffers from a reduced data cache hit rate. The work presented in this thesis does not compromise the cache performance.

Balasubramonian [Bal04] advocates the use an address predictor for load and store instructions and a dependency predictor for stores to speculatively execute load instructions and improve the performance of clustered microarchitectures for selected SPECfp benchmarks. Furthermore, he argues that broadcasts of store addresses between clusters delay

the disambiguation of load instructions and pose a critical performance bottleneck for distributed cache organizations. Our own experiments (see Section 4.2.3) confirm the poor performance of configurations with a conservative issue policy. They also suggest that good results can be achieved across all SPECint and SPECfp benchmarks by issuing load instructions speculatively. We find that speculative issue effectively solves the problem of delayed load disambiguation.

Irie et al. [Iri05] and Watanabe et al. [Wat05] propose to adapt memory bypassing [Mos97b] for clustered microarchitectures. Their proposals are orthogonal to those presented in this thesis. However, memory-bypassing techniques provide little performance benefit over simpler memory dependence prediction, which guides the scheduling of speculative execution of load instructions. [Loh02] For this reason we include memory dependence prediction but not memory bypassing in our architecture proposal. (See Section 4.2.3.)

Gunadi and Lipasti [Gun07] propose a scheme with late allocation and an unordered store queue for a traditional architecture with a single centralized memory unit. They adopt solutions from the realm of non-compacting issue queues to unordered store queues. This allows them to solve the problem of how to select instructions by age in an unordered queue. Their circuit could be used in the context of this thesis substituting the selection mechanisms described in Sections 2.2.4, 4.1.9 and 4.1.11.

Many more academic proposals exist, which do not focus on—but can be applied to—distributed memory units. Many of these proposals are orthogonal to the techniques presented in this thesis. The previous paragraph already mentioned memory bypassing. Memory bypassing and related schemes [Tys97, Mos97b, Rei98, Mos99, Par03, Sha05, Sha06, Sub06a] predict memory dependencies and pass results between instructions based on these predictions. These schemes allow alternative organizations of the memory unit but have little impact on performance. [Loh02] In this thesis, we use memory dependence prediction to guide the speculative issue of load instructions. [Mos97a, Kes98, Chr98, Ond99, Mos00, Fan06, Sub06b] Our experiments indicate that speculative issue provides an important performance boost over a conservative issue policy (see Section 4.2.3). Some publications propose to filter searches for memory dependencies to limit the usage of content addressable memory (or to limit the re-execution of load instructions). [Set03, Cai04, Rot05, Cas06] While we do not explicitly cover search filters in this thesis, our distributed memory unit could be extended to include search filters. Finally, there are proposals to organize load and store queues using concepts known from caches. [Sto05][Gan05][Gar06] Cache–like organizations are especially attractive for large queues [Gan05] but a distributed memory unit naturally leads to multiple smaller queues. Therefore, we do not adapt a cache–like organization but instead propose unordered load–store queues (see Section 4.1.4), which are more suited for a distributed organization.

2.4 Bank Prediction

The bank predictor is a key element in the design of a disambiguation unit, which is distributed by banks. The predictor makes the bank accessed by a memory instruction available in an early stage of the pipeline and thus allows the architecture to plan the allocation of resources accordingly, minimize communication delays, and simplify the overall microarchitecture. It is therefore important to estimate the cost of the predictor as well as its accuracy. These two parameters ultimately decide first, if it is reasonable at all to employ a bank predictor, and second how to make use of the bank predictions. E.g. a bank predictor with near 100% accuracy allows aggressive speculation and simplification of the microarchitecture, even if the misprediction penalty is considerable. An accuracy significantly lower than 100% on the other hand suggests that the prediction is better used as a hint to improve resource allocation and reduce communication delays, but to avoid excessive speculation. The resulting design of the microarchitecture does depend in no small degree on the characteristics of the bank predictor.

There do not exist many publications on bank predictors. We are only aware of two publications by other authors, the already mentioned paper by Yoaz et al. [Yoa99] and a publication by Neefs et al. [Nee00]. Yoaz et al. primarily discuss configurations with two cache banks where the predictor produces two single bit results (one bit to indicate the bank and another bit to indicate the confidence in the prediction). Consequently, they propose designs based on binary predictors, like branch predictors. To extend their results to more than two banks they suggest two approaches. First to predict multiple bits independently from each other using a different predictor for each bit, and second to use known address predictors. None of these approaches is discussed in detail in their paper, which instead focuses on configurations of two cache banks.

Neefs et al. [Nee00] evaluate bank predictors based on designs for value predictors and present very detailed results e.g. on the accuracy obtained for various address bits. While they demonstrate that bank predictors are a useful concept, their proposals for implementation remain very abstract. In fact, they do not discuss the cost of the predictors, many of whom assume near–infinite resources.

In this work, we will study bank predictors in greater detail and pay special attention to configurations with a reasonable implementation complexity. Based on the results of that study we will go on to propose a microarchitecture for a distributed memory unit, taking into account the characteristics of the bank predictors.

CHAPTER 3

BANK PREDICTORS

Bank predictors are a key element in the design of our distributed memory unit. They allow the microarchitecture to establish a mapping of memory instructions to clusters very early in the pipeline long before the address of a memory access is known. This knowledge is used to maximize locality by assigning memory instructions to those clusters where their corresponding memory locations are (most likely) mapped.

Especially the accuracy of the bank predictor is an important design goal. It determines how much effort on part of the microarchitecture is necessary to handle bank mispredictions. In this chapter, we will apply the concepts of predictors from other domains to the problem of bank prediction with special focus on the quantitative metrics of implementation cost and accuracy.

3.1 Introduction

The most popular and by far the best-studied predictors are branch predictors. Binary predictors are also frequently used in memory units e.g. to predict memory latency, memory dependencies etc. Value predictors (especially address predictors, which can be considered a type of value predictors) are another important class of predictors. Many of the techniques used for branch and value prediction can be applied to bank prediction. There are however some differences. While branch predictors predict binary values and value predictors typically

predict 32 or 64–bit quantities, bank predictors predict values of a relatively small range depending on the cache configuration.

Branch predictors often make use of the binary nature of their predictions. Examples are counters that are often employed in binary predictors. The two binary states can be mapped to an increment or a decrement action; for states that are more complex, there is no simple extension. Other examples are voting predictors. For binary predictors, three predictors are always sufficient to obtain a majority. If more complex states are predicted, the number of required predictors quickly becomes unreasonable.

3.2 Methodology of the Evaluation

We use traces to evaluate the performance of the different bank predictors. The traces contain the Program Counter and the address of all memory instructions. Because the bank prediction is performed during an early pipeline stage it is common for many memory instructions to be predicted before any of these predictions can be verified and fed back to the predictor to update it accordingly. After the first experiments, we noticed the importance of the delay between the bank prediction and the calculation of the address. This delay has a significant effect on the results of the predictor. Therefore, we reconstructed the traces to include the delay of the predictor updates.

To generate the traces we used the SimpleScalar 3.0 toolset [Aus02] with the Digital Alpha instruction set architecture. We generated one trace for each SPECint2000 benchmark except for 254.gap because the simulator could not execute the benchmark correctly at the time we generated the traces. Each trace consists of the first 100 million memory references (loads and stores) after skipping the initialization phase of each benchmark[13]. We simulate a generic 8–way superscalar architecture with a delay of at least 26 cycles between the pipeline stages where the bank is predicted and where the bank predictor can be updated.

We use a configuration of eight cache banks, which are interleaved by 64–bit words. Therefore, we use the bits three, four and five of an address to determine the cache bank.

3.3 Description and Evaluation of the Predictors

3.3.1 Last Bank Predictor

This simple predictor consists of a single table like the bimodal branch predictor. [Smi81] However, unlike the bimodal branch predictor each entry of the table does not consist of a counter but of a small word that represents a cache bank. In our case, each word contains three bits to address eight cache banks. We experimented with saturating counters and assigned a counter to each of the three bits but results were discouraging. By splitting the

[13] The initialization phase of each benchmark was determined by inspecting and instrumenting the source code of each benchmark.

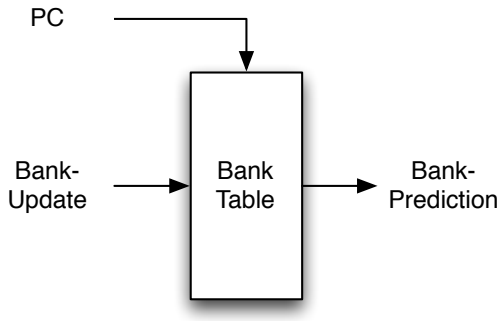

Figure 3.1: Last Bank Predictor.

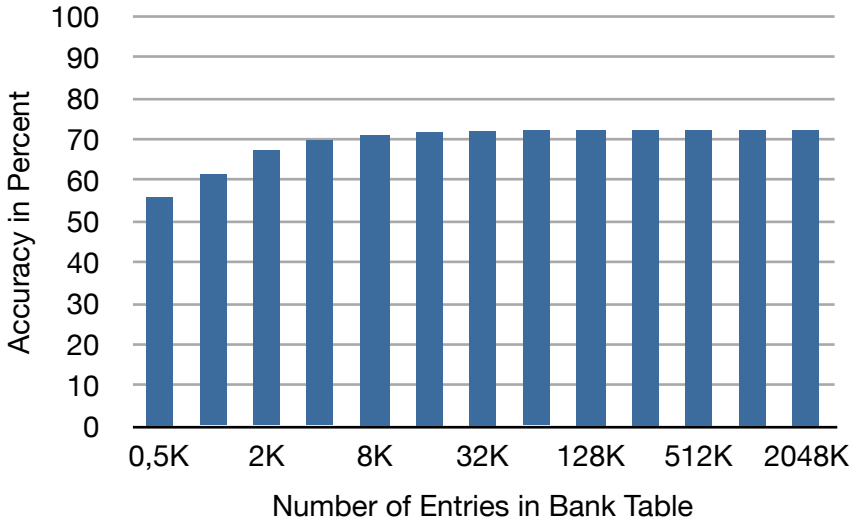

Figure 3.2: Accuracy of the Last Bank Predictor.

word into independent bits, the correlation between the bits, and therefore important information, is lost.

Figure 3.1 shows the structure of the predictor. The Bank Table is indexed by the least–significant bits of the Program Counter, ignoring the two least–significant bits, which have no function in a RISC architecture with 32–bit instruction words. The least–significant bits tend to contain more entropy than the most–significant bits.

In the absence of aliasing, which occurs when two or more different instructions are mapped to the same table entry, this predictor always predicts that an instruction will again access the last bank it accessed. This assumption is reasonable for references to global variables and in many cases for accesses to the stack and the heap. Even in the case of destructive aliasing (which occurs when aliasing instructions access different banks) this predictor recovers rapidly. However, due to its simple structure it is not able to capture more complex access patterns.

The bank update and the bank prediction designated in Figure 3.1 do not occur simultaneously. Instead, the update is delayed by several cycles as described above in

Section 3.1. This delay can affect results if the same static instruction enters the pipeline multiple times e.g. in a very small loop.

Figure 3.2 shows the accuracy of the Last Bank predictor as a function of Bank Table size. The maximum accuracy of 72.3% is reached at 128K entries (48Kbytes), more than 70% accuracy are already reached with a table of 8K entries (3Kbytes). The accuracy reaches a limit at great table sizes when the number of static memory instructions is significantly lower than the number of entries in the Bank Table. Any further table enlargement results only in more entries that are unused. The remaining 27.7% of all memory instructions exhibit access patterns that are more complex and do not always access the same cache bank.

3.3.2 Global Bank Predictor

This kind of bank predictor is inspired by global branch predictors. The most recent accessed banks are kept in a single shift register. When a new result is shifted into the register, the oldest one is discarded. The contents of this shift register are called global history because its results may originate from different static instructions. The global history forms the index into the Bank Table that contains the next prediction. The size of the global history is defined by the size of the Bank Table. Figure 3.3 shows the structure of the predictor.

This predictor gave good results in our first experiments, where we assumed that the predictor updates occur immediately after the prediction. The predictor could effectively identify instructions based on their history. However, when we adapted more realistic, delayed predictor updates, the results of this predictor became unsatisfactory. Because the delay between prediction and update depends on the complex out-of-order pipeline, it became difficult for the predictor to identify instructions solely by the outdated global history.

To solve this problem, we implemented speculative updates of the global history. After each prediction, the global history is updated speculatively with the previously predicted value, while the Bank Table is left unmodified. Speculatively modifying the Bank Table would be difficult to undo, and the global history is sufficient to identify instructions. During the non-speculative update both, the history and the Bank Table, are updated. To update the history in case of a misprediction, the incorrectly inserted bits are replaced with the correct bits. In this respect, the bank predictor differs from branch predictors, which also perform a speculative update of global history registers, because mispredicted branches empty the whole pipeline and most of the global branch history. Bank mispredictions are easier to recover from, as they typically incur an extra delay but not a complete pipeline restart.

The speculative update does indeed improve the results of the Global Predictor but results, especially for small table sizes, remain poor. Small tables are restricted to a small bank history, which may not be enough to predict future behavior. Another reason are updates to the Bank Table that are not performed with the right global history index. Over time, instructions can be identified by slightly incorrect global histories too, but this requires more

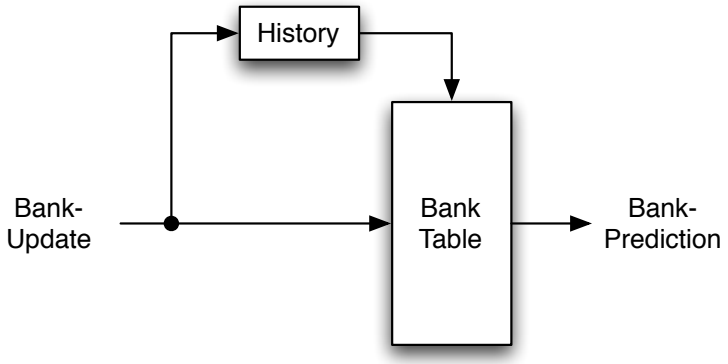

Figure 3.3: Global Bank Predictor.

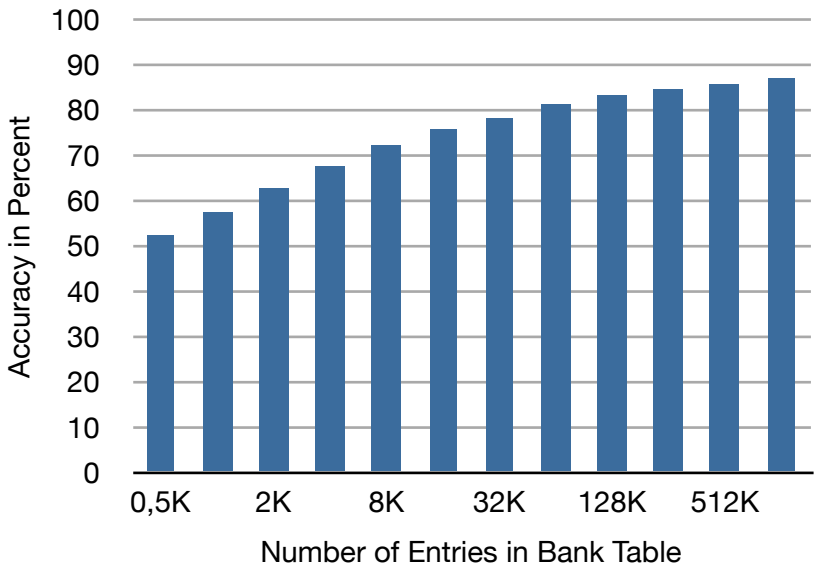

Figure 3.4: Accuracy of the Global Bank Predictor using speculative updates.

Bank Table entries and more time to update the Bank Table with all the incorrect global histories. The increased use of table entries leads to more aliasing. Figure 3.4 shows that results for small tables are inferior to the Last Bank Predictor however the Global Predictor improves with table size and is able to beat the Last Bank Predictor for tables with more than 16K entries (6Kbytes). However, the Global Predictor stagnates well below an accuracy of 90% even for huge tables.

3.3.3 Gshare Bank Predictor

The Gshare branch predictor was proposed by McFarling [McF93]. Our adoption to bank prediction is similar to the Last Bank Predictor and the Global Bank Predictor. The index into the Bank Table is formed by an XOR of the global history and the Program Counter. Each entry of the Bank Table contains a bank prediction. The Program Counter is used exactly as it is used for the Last Bank Predictor. The table size determines how many bits are extracted from the Program Counter. The two least significant bits of the Program Counter are always

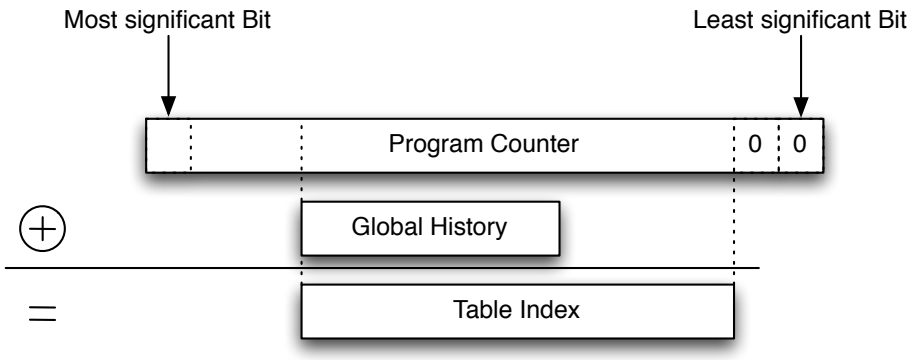

Figure 3.5: Calculation of index for Gshare Bank Predictor. The two least significant Program Counter bits are always zero and are therefore not used. Likewise the most significant bits of the Program Counter are not used in the calculation.

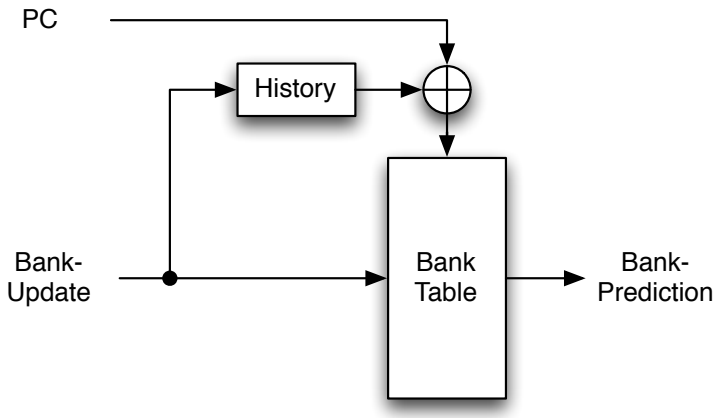

Figure 3.6: Gshare Bank Predictor.

zero and are therefore not used. The global history is shorter than the table size. In other words, the global history contributes fewer bits to the index than the Program Counter. This raises the question, which bits of the Program Counter should be XORed to the global history. Because the least significant bits of the Program Counter contain more entropy than the most significant bits, the entropy of the whole index is maximized if we combine the Program Counter in the way illustrated by Figure 3.5. We were able to confirm experimentally that this alignment gives the best results. Figure 3.6 shows the overall structure of a Gshare Bank Predictor.

The structure of the Gshare Bank Predictor allows a variable history size. We experimentally determined the best history size for each table size. Figure 3.7 shows how the accuracy is affected by the size of the history. A history size of zero is a special case that results in a Last Bank Predictor. A history of a single bank is enough to clearly outperform the Last Bank Predictor for all table sizes. The Gshare Bank Predictor also outperforms the Global Bank Predictor for all table sizes. This is due to the Gshare Bank Predictor's ability to identify static instructions using bits of the Program Counter in its index. The figure further shows that greater table sizes favor larger histories. This is reasonable since a larger history

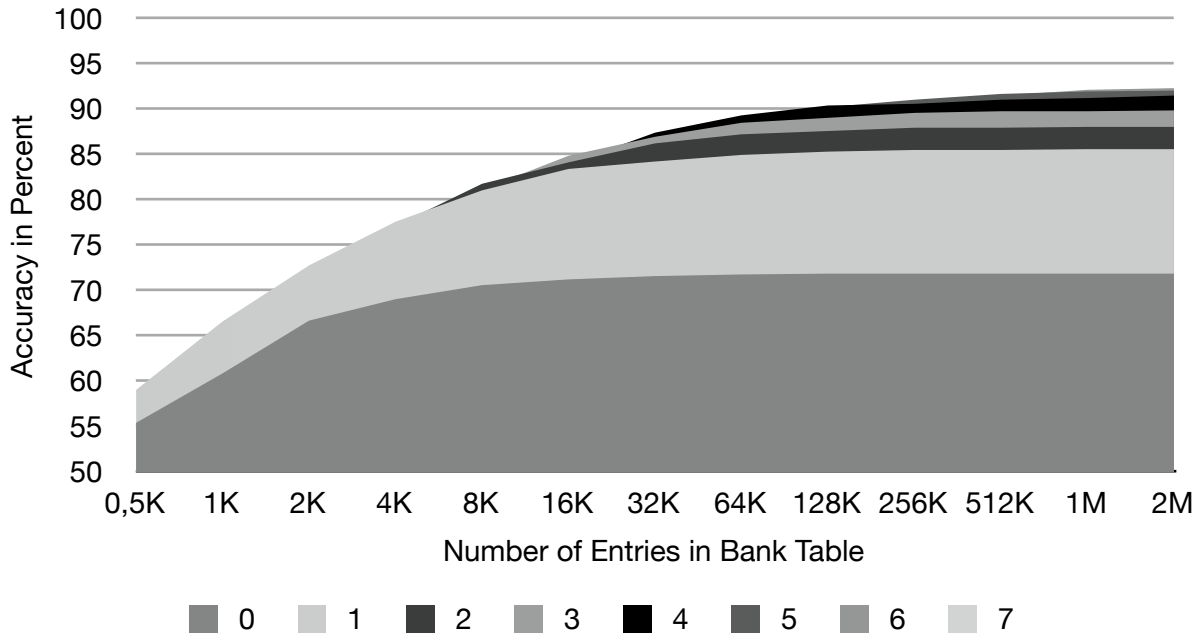

Figure 3.7: Accuracy of Gshare as a function of history and table size. The history size in three–bit entries is shown as color, smaller histories starting from below.

Table Size	1..64	128..4K	8K	16K	32K..128K	256K..512K	1M..2M
History Size	0	1	2	3	4	5	6

Table 3.1: Optimal history size for Gshare Bank Predictor. Table size as well as history size are given in entires. Each entry references one of eight banks and occupies three bits. E.g. the optimal history size for a table of 16K entries is three three–bit entries for a total nine bits of history.

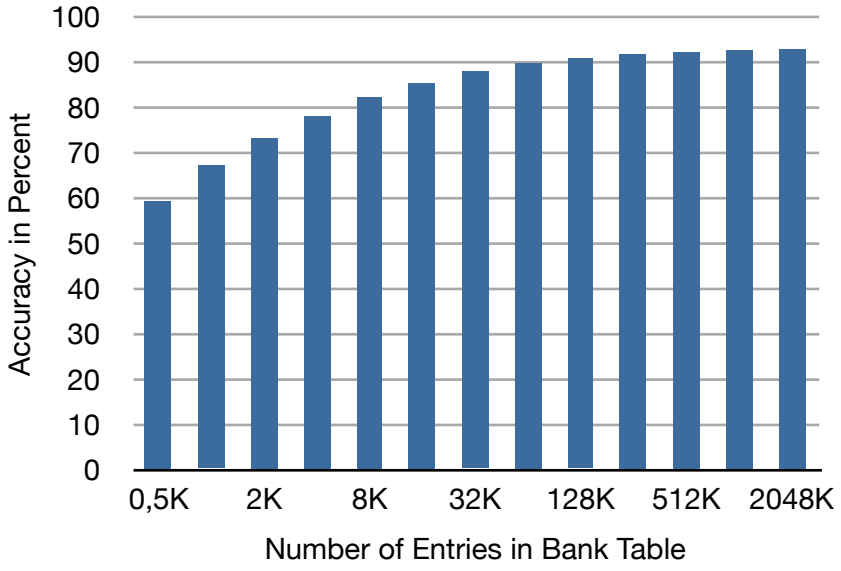

Figure 3.8: Accuracy of the Gshare Bank Predictor with optimal history size.

allows the predictor to effectively address more table entries for each static memory instruction.

Since it is hard to read the exact numbers in Figure 3.7 we show the optimal history sizes again in Table 3.1 and the accuracy of the optimal history for each table size again in Figure 3.8. In our experiments with an unrealistic immediate update the optimal history size was significantly larger. This indicates that larger delays lead to lower optimal history lengths.

Figure 3.8 shows that the accuracy of the Gshare Bank Predictor increases, but above 512K entries there is hardly any improvement, and the accuracy reaches a plateau at 93%. The remaining 7% are instructions with complex patterns that the structure of Gshare cannot capture and, to a small degree, static instructions that are predicted for the very first time when the predictor has not yet learned to predict their access patterns.

3.3.4 Local History Bank Predictor

Local history predictors [Yeh92] have been used for both branch prediction and value prediction. In the context of branch prediction, a local predictor is similar to a global predictor except that it uses a table of history registers instead of a global history register. This table is indexed with the Program Counter of the branch instruction.

In the realm of value prediction local history branch predictors are known as finite context method predictors. These predictors are described by Sazeides and Smith. [Saz97] Unlike local history branch predictors they hash the history to form the index for the Bank Table.

Because bank identifiers consist of only a few bits, hashing does not improve the accuracy of a Local History Bank Predictor, as we have experimentally confirmed. We have also observed that including other information—like parts of the Program Counter—in the computation of the Bank Table index [Bal02] is counterproductive, as it diminishes the constructive aliasing [You95] between identical history patterns. See Figure 3.11 for quantitative data.

Since this predictor contains two tables, there are several possibilities to split the available transistor budget between them. We found that this division has little effect on prediction accuracy. A history table that has eight times as many entries as the Bank Table generates the best results among the alternatives that we have explored. In general, it is preferable for small predictors to have bigger history tables and for large predictors to have bigger Bank Tables. The reason is that conflicts in the history table are more severe than a short history (which is implied by a small Bank Table), but once the history table is big enough to hold the memory instructions of the current working set, the remaining storage is better invested in a longer history and hence a larger Bank Table. However, the resulting

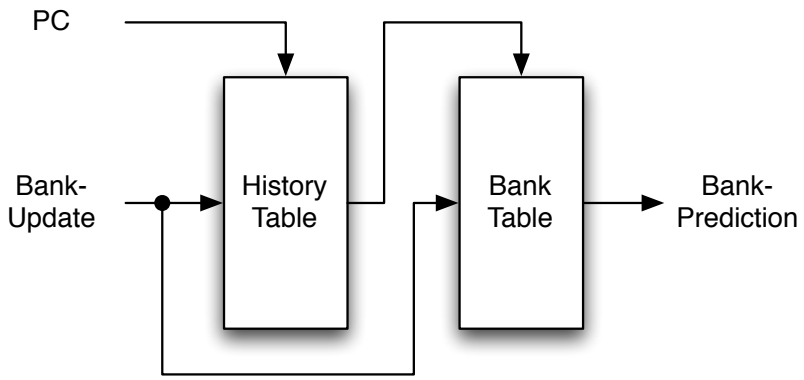

Figure 3.9: Local History Bank Predictor.

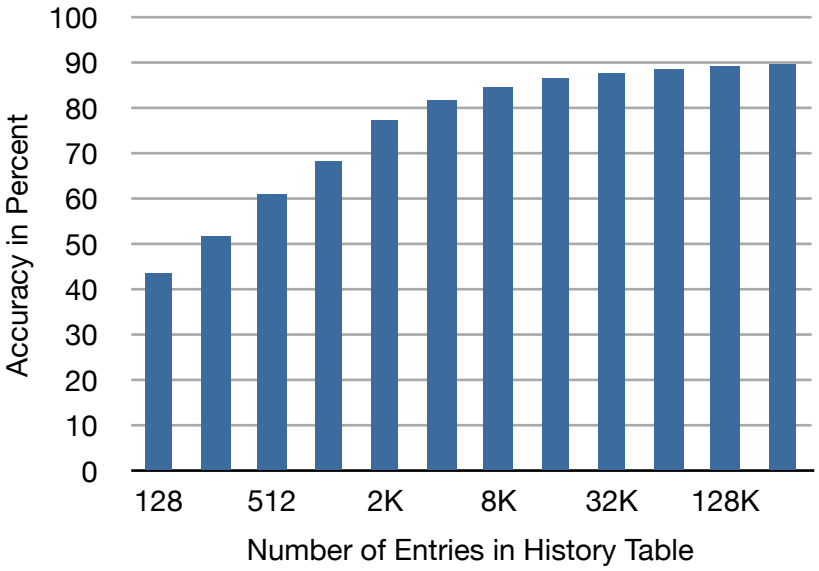

Figure 3.10: Accuracy of the Local History Bank Predictor. In the configuration shown the history table is eight times greater than the Bank Table.

difference in accuracy is so small that for simplicity we settle on a fixed factor for the relative table sizes.

Figure 3.9 summarizes the structure of the Local History Bank Predictor. The size of the entries in the history table is defined by the size of the Bank Table. We also evaluate configurations where the history size, measured in bits, is not an exact multiple of three such that the history can contain partial banks. This has no negative effects on the accuracy.

Figure 3.10 shows the accuracy of the Local History Bank Predictor as a function of table size. The accuracy rises significantly up to a size of 2K entries for the history and 256 entries for the Bank Table. Afterwards the accuracy increases more slowly until it reaches a plateau around 90% of accuracy. Later we will see that a Gshare Bank Predictor has a higher accuracy than a Local History Bank Predictor does if we compare configurations with a similar bit budget. On the same basis of comparison, the Local History Bank Predictor compares favorably with the Last Bank and the Global History Bank Predictor.

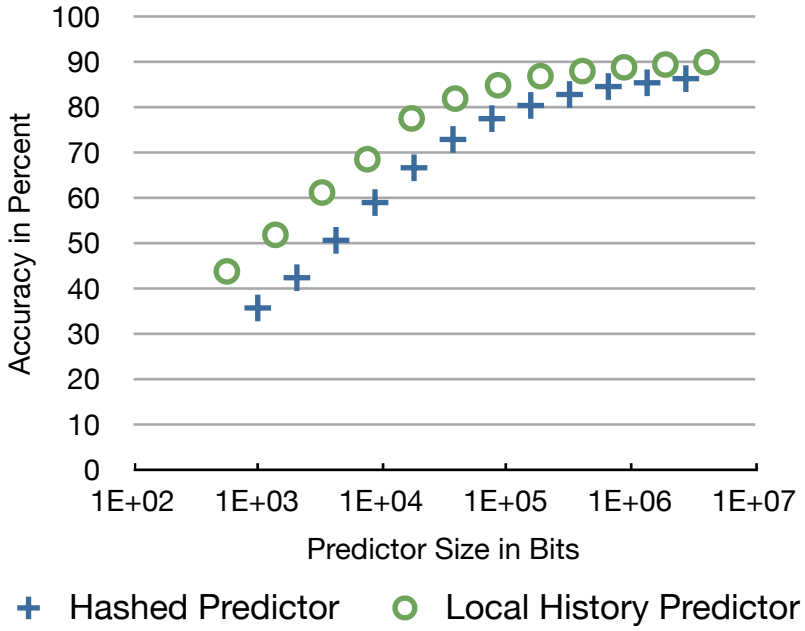

Figure 3.11: Comparison of Hashed and Local History Predictor. Accuracy is shown as a function of the predictor size in bits for the Local History Bank Predictor presented in this section and for another proposal. [Bal02]

Local predictors are more efficient than global predictors are when the bank accessed by an instruction does not depend on its immediate predecessors. Notably constructive aliasing in the Bank Table occurs much more frequently than in the case of global predictors.

Balasubramonian et al. [Bal02] proposed a predictor that is closely related to the Local History Bank Predictor but differs in two aspects. First, the index into the Bank Table does not only consist of bits from the history table. Instead, the index is formed by a hash function (a simple XOR) of the history bits and two bits from the Program Counter. Second, the entries of the Bank Table are bimodal two–bit counters instead of the simpler entries in our proposal.

In Figure 3.11 we compare the Local History Bank Predictor with the proposal of Balasubramonian et al. [Bal02] Because the resulting Bank Table entries have a different size we compare the predictors by their overall size in bits. As can be seen in Figure 3.11, our implementation of their proposal shows inferior accuracy than a plain Local History Bank Predictor.

3.3.5 Stride Predictors

Stride predictors compute the predicted result, rather than obtaining it directly from a previously observed pattern. Stride predictors are based on the observation that the difference (or stride) between two successive values generated by the same instruction is often a constant. Instead of directly predicting the result, they predict this constant. The final

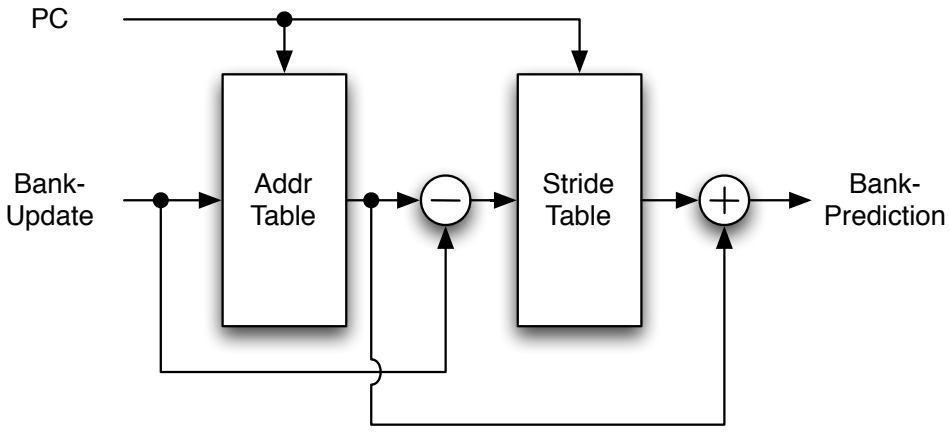

Figure 3.12: Stride Predictor.

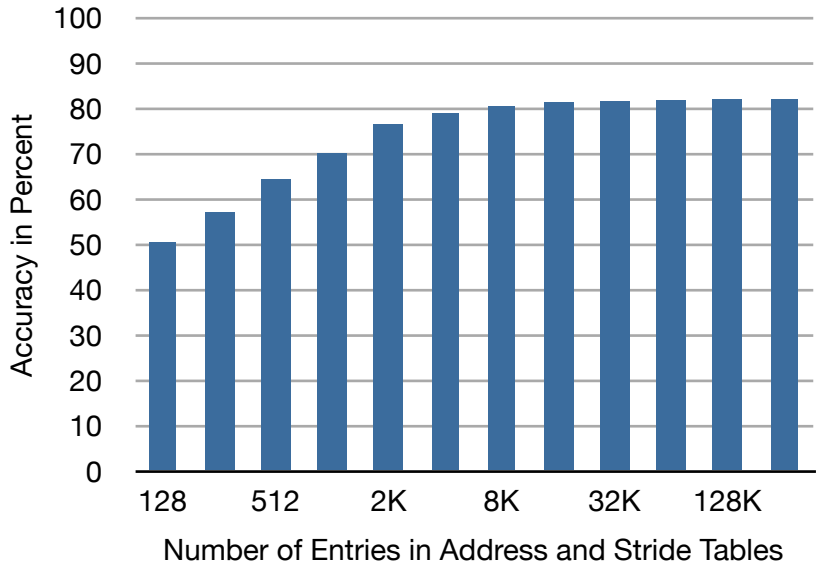

Figure 3.13: Accuracy of the Stride Predictor.

prediction is then generated in a second step by adding the last encountered value to the predicted stride.

Figure 3.12 shows the basic structure of a Stride Predictor. To make the algorithm more robust against occasional mispredictions we use the two–delta method [Eic93]: a stride (or delta) must occur at least two times in a row before it is used in future predictions. The two–delta method requires an additional table to store the last encountered stride. When the predictor is updated, it compares the current stride with the last stride from this table. If the two strides match, the predictor updates the regular stride table. In total, there are three tables: an address table, a last stride table, and a regular stride table. For simplicity, we omit the last stride table in Figure 3.12.

If used as a bank predictor, it is not necessary to deal with the full addresses and strides. Instead, only the address bits that are used to select the bank and all lower significant bits are

included. Including the lower significant bits has the benefit that strides as small as one byte can be correctly predicted. However, this doubles the bit budget for a fixed table size.

Figure 3.13 shows the accuracy of the stride predictor as a function of the size of the history and stride tables. At about 16K table entries the accuracy reaches its plateau. This is similar to the Last Bank predictor, which reaches its plateau at the same number of table entries. The reason is that after all static instructions are mapped to a different table entry; any more entries that are added remain unused and do not affect the results of the predictor. However, the plateau of the stride predictor at about 82% is 10% higher than plateau of the Last Bank Predictor, but comes at a much higher cost per table entry.

3.3.6 Local Stride Predictor

The Local Stride Predictor is a combination of the Local History Predictor with the Stride Predictor. It is based on the Differential Finite Context Method Value Predictor [Goe01].

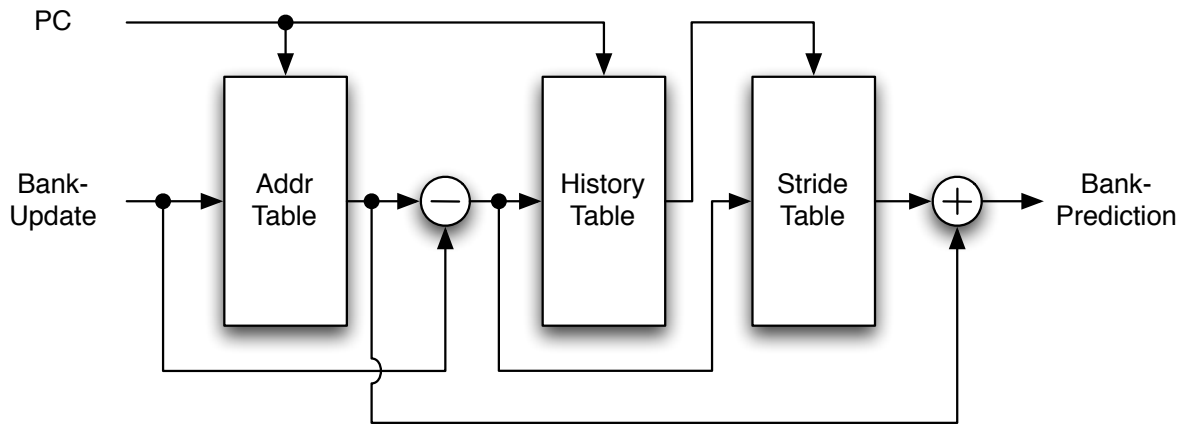

Figure 3.14: Local Stride Predictor.

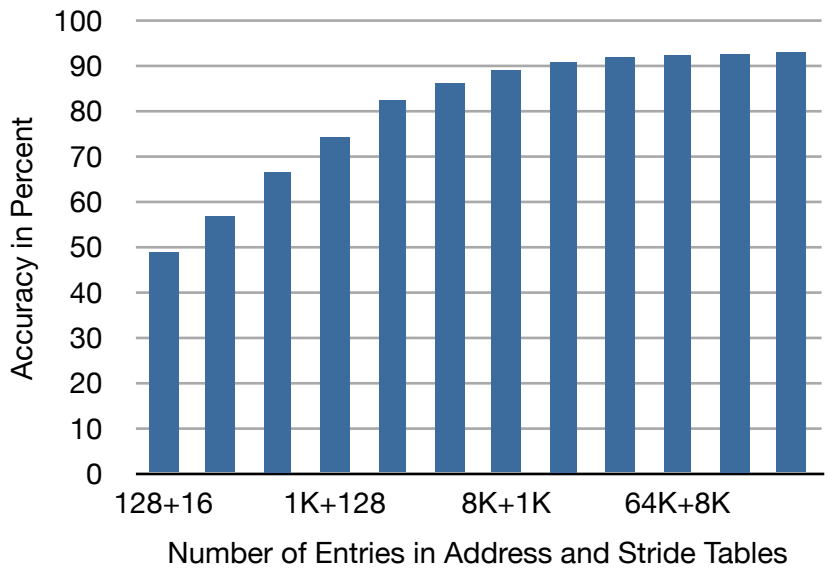

Figure 3.15: Accuracy of the Local Stride Predictor.

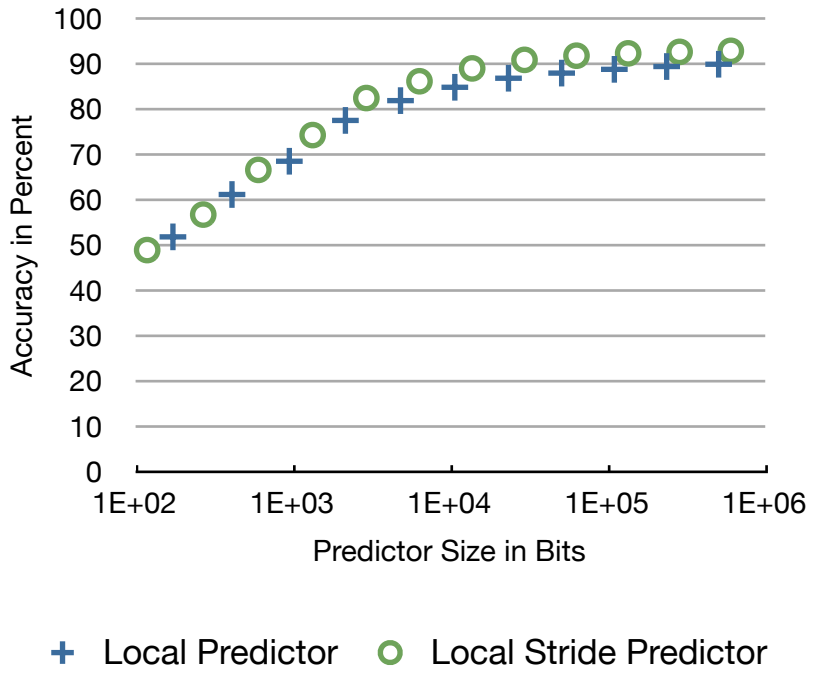

Figure 3.16: Comparison of Local Predictor and Local Stride Predictor.

Figure 3.14 summarizes the structure of this predictor. Similar to the Local History Predictor it possesses a history table of shift registers that is indexed by the Program Counter. However, unlike the Local History Predictor the histories contain strides instead of banks, consequently the Bank Table of the Local History Predictor is also substituted by a stride table. To calculate the strides an address table is required. However, unlike the Stride Predictor, this predictor does not use the two–delta method. This structure is attractive because it aims at constructive aliasing in the stride table: i.e. static instructions that always access the same bank generate a constant history of zero in the history table and a stride of zero in the first entry of the stride table. Like for the Local History Bank Predictor, we choose a fixed factor of eight to determine the relative sizes of the tables. Therefore, the history table has eight times more entries than the stride table. Address and history tables have the same number of entries.

Figure 3.15 shows the accuracy of the Local Stride Predictor. For larger predictors it reaches 92.7%–a higher plateau than that of the Local History Predictor (compare Figure 3.10). This comes at the cost of the address table, which is absent in the Local History Predictor. Figure 3.16 compares the two predictors considering their respective sizes. The Local Stride Predictor is also more attractive than the Local Predictor is if we take cost into account.

3.3.7 Gskew Bank Predictor

The Gskew Bank Predictor is modeled after the Gskew Branch Predictor described by Michaud et al. [Mic97] It is a voting predictor that consists of three predictors. These smaller predictors are similar to Gshare predictors but use each a different specially designed index

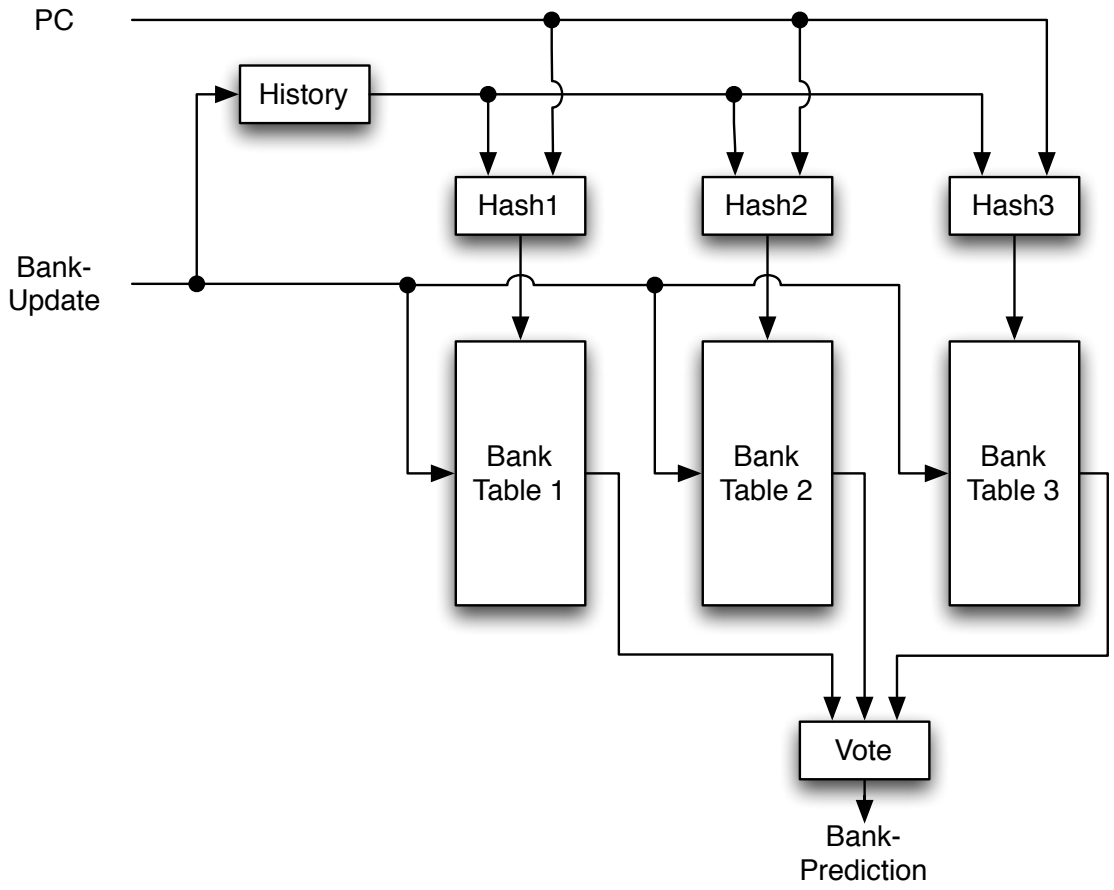

Figure 3.17: Gskew Predictor.

Table Size	1..256	512..2K	4K..16K	32K..128K
History Size	1	2	5	6

Table 3.2: Optimal history size for Gskew Bank Predictor. E.g. the optimal history size for a table of 16K entries is five banks (15 index bits).

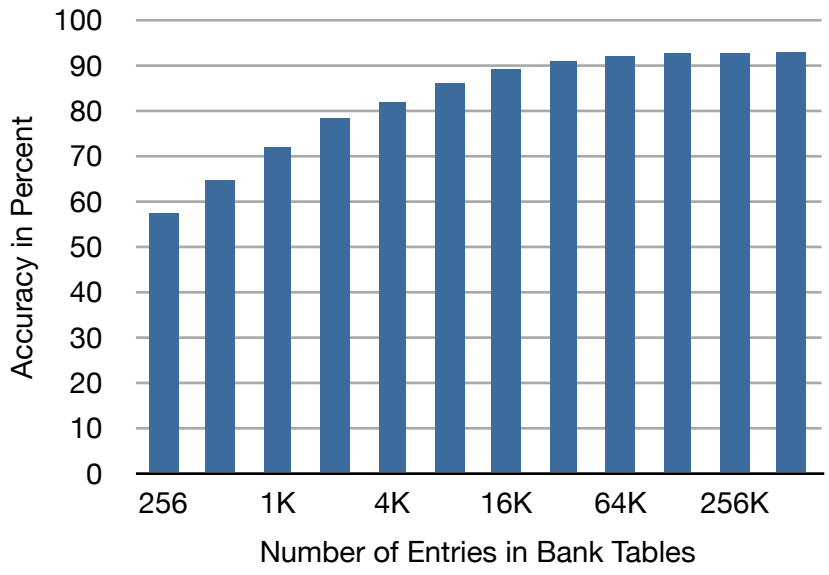

Figure 3.18: Accuracy of the Gskew Bank Predictor. The history size for each table size is chosen to optimize overall accuracy for the SPECint benchmarks.

function. These index functions have the notable property that if two inputs do produce a conflict in two tables, they are guaranteed not to do so in the third table. In this way, Gskew minimizes aliasing conflicts. Adapting these predictors to bank prediction is straightforward. However, the voting function must be modified, because bank predictors are not (necessarily) binary and situations can arise where all predictors disagree with each other and there is no majority for any single prediction. Since there is no number of predictors that can guarantee a majority in all cases, we designate a master predictor who overrules the other predictors if a tie occurs. Figure 3.17 shows the structure of the Gskew predictor.

The predictors use a global history that is combined with the Program Counter to form the index into the Bank Table. Therefore, the problem of the optimal history length arises similarly to the Gshare Bank Predictor. We found the history size shown in Table 3.2 to deliver the best overall accuracy for the SPECint2000 benchmarks.

We did further experiments with variations of gskew, e.g. we implemented e–gskew [Mic97] and 2bcgskew [Sez99] predictors, however these predictors did not give better results than a plain gskew predictor. We found that a partial update strategy gives slightly better results than a total update strategy. With a total update strategy, the predictor is always updated, with a partial update strategy, it is not updated if the prediction was correct, even if one of the predictors gave an incorrect answer.

Figure 3.18 shows the results of the Gskew Bank Predictor as a function of the size of a single table. Results are very good but it must be noted that the predictor uses three tables instead of one (the x–axis show the size of a single table). The Gskew Predictor reaches a plateau at about 128K entries and an accuracy of 93%.

3.3.8 Tournament Predictor

The Tournament Predictor [McF93] consists of three predictors: two bank predictors and a meta predictor that is used to choose the result from one of the bank predictors. Because it contains two different types of bank predictors, a Tournament Predictor can predict the sum of patterns that each of the bank predictors is able to predict. This flexibility out-weights in many cases the additional overhead of the meta predictor.

In this section, we examine a combination of a Local History Predictor and a Gshare Predictor (the same combination as proposed in [McF93]). We tried other combinations of bank predictors but of all the variants we evaluated this combination gave the best results.

The meta predictor is updated only if exactly one of the predictors gives a correct answer but the other does not. Because the meta predictor is a binary predictor it is convenient to use saturated counters. We use relatively small two–bit counters to allow the meta predictor to alternate quickly between the bank predictors. There are two ways to update the bank predictors. One is to always update all bank predictors as if they would operate independently

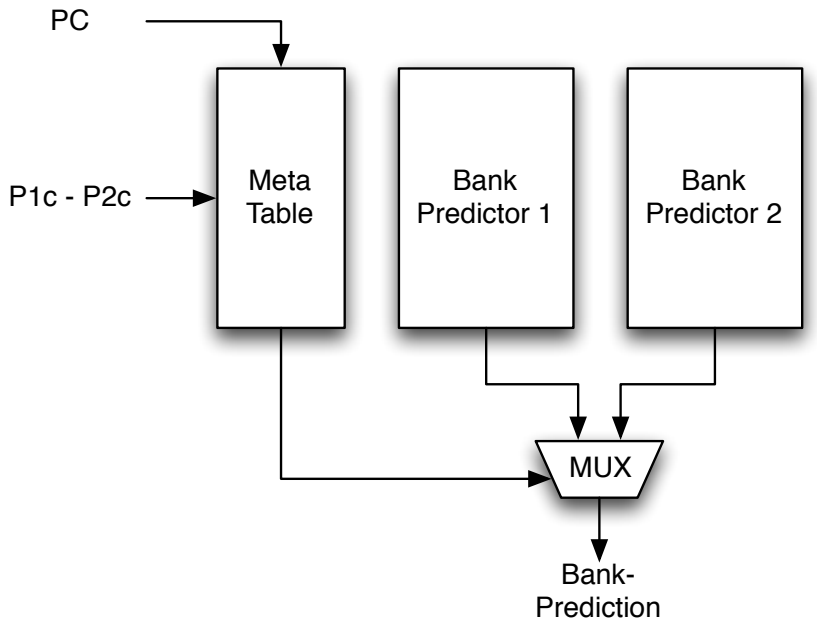

Figure 3.19: Tournament Bank Predictor.

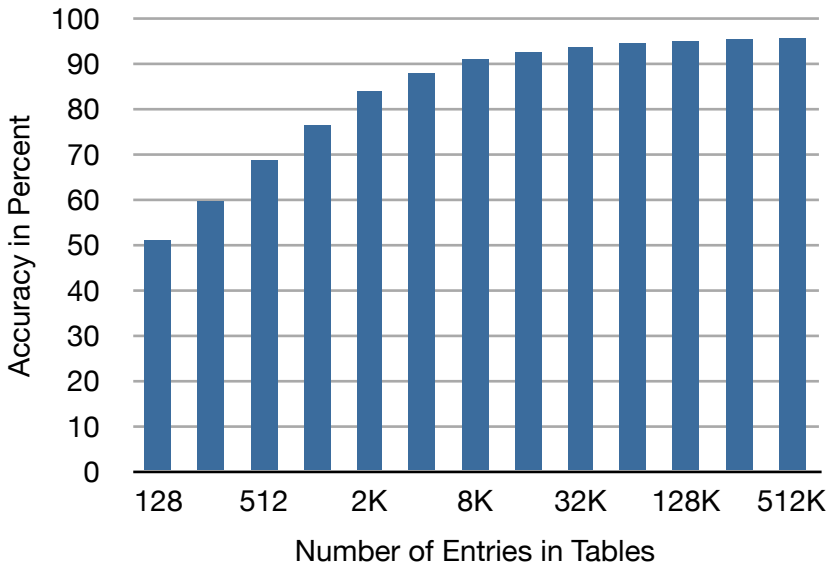

Figure 3.20: Accuracy of the Tournament Predictor. This predictor combines a Gshare and a Local History Predictor. All tables of the predictor are of the same size except the Bank Table of the Local History Predictor, which is eight times smaller.

(total update). The other is to update the bank predictors only if the overall prediction is wrong (partial update). We found that the total update strategy gave slightly better results.

Figure 3.19 shows the basic structure of the predictor. The signals P1c and P2c are set if the last prediction of bank predictor 1 and predictor 2 respectively was correct. As described above the saturated counter of the meta table that is indexed by the Program Counter is updated with the difference of the two signals. The bank predictors 1 and 2 operate as if they were independent. For simplicity, we do not show their input and output signals here.

Figure 3.20 shows the accuracy of the Tournament Predictor. The results especially for the larger sizes are the best so far. The accuracy reaches a plateau at about 64K entries at 96% accuracy. However, to compare the results for small configurations we have to remember that this predictor uses four tables and is more complex than other predictors with the same table size.

3.4 Comparison of Bank Predictors Based on Bit Budget

Figure 3.21 shows the accuracy of all predictors as a function of size. While the figure is quite crowded the most important result is clear to see: Only two predictors—Gshare and Tournament—offer the best combination of accuracy and size for the whole range of sizes. Up to about a size of 3Kbytes, Gshare is the best option; for bigger predictors the Tournament Predictor is best.

Simple predictors with a single table like Gshare and Last Bank excel at lower sizes where the more complex predictors have to work with smaller tables and therefore experience more aliasing. This disadvantage disappears with larger sizes. The Tournament Predictor is the most flexible because it can make use of local as well as global history. In the absence of

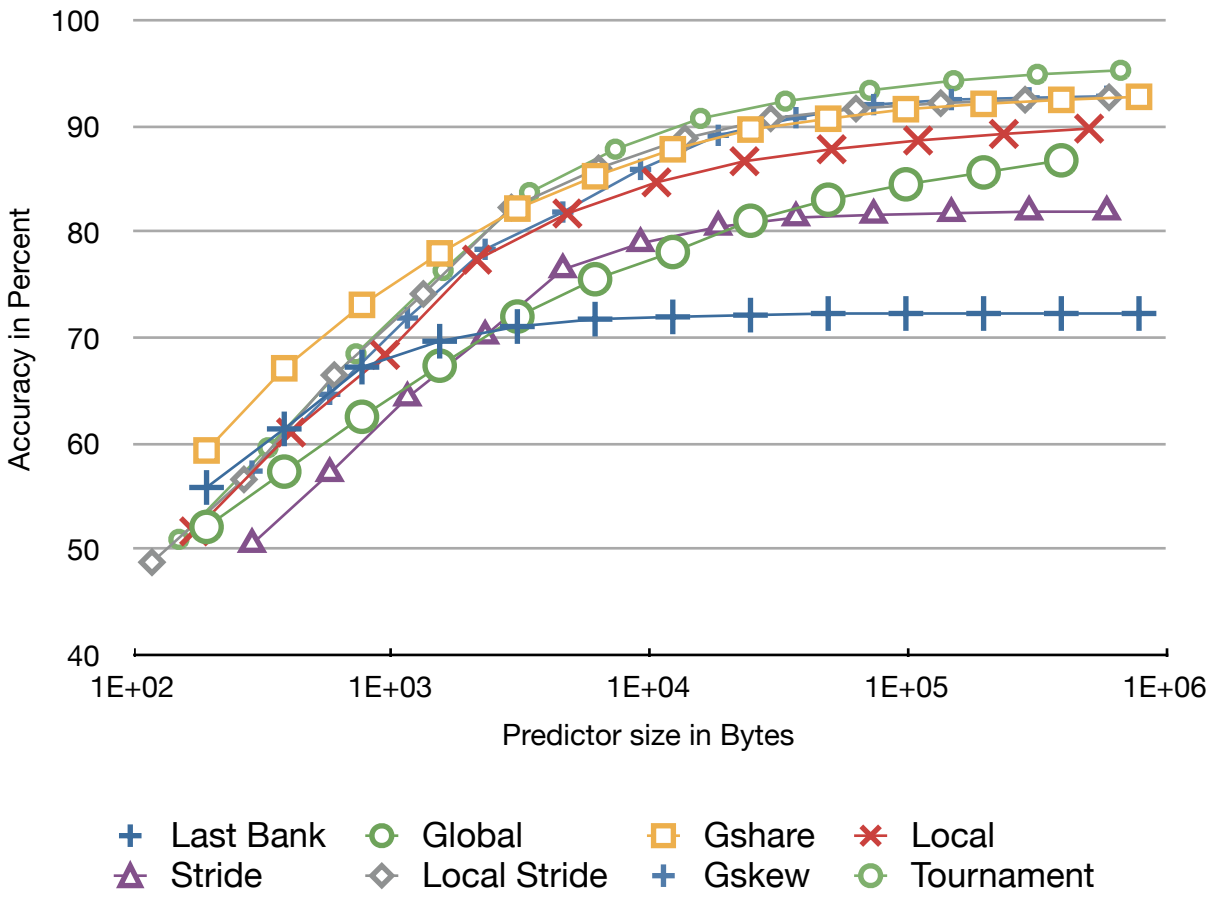

Figure 3.21: Accuracy of all predictors as a function of the predictor size. Some predictors use the same symbols: Last Bank uses bigger crosses than Gskew and Global uses bigger circles than Tournament.

aliasing that bigger predictors offer, this flexibility results in the best accuracy. Both of the two best predictors are based on branch predictors.

3.5 Accuracy and Prediction Rate

In this chapter, we have characterized the different bank predictors by their size and by their accuracy. But some applications of bank predictors require an additional parameter: confidence. This parameter describes how likely a prediction is correct. The confidence is usually a binary parameter. If a particular prediction has no confidence, it is unused in most applications. The accuracy in this case refers only to confident predictions. The fraction of all confident predictions is called the prediction rate.

The two most relevant proposals to our work from Yoaz et al. [Yoa99] and from Zyuban and Kogge [Zyu00][Zyu01] rely on hardware simplifications, which seriously penalize bank mispredictions. Consequently, these proposals require an accuracy very close to 100%. Yoaz et al. describe predictors for two banks. In our experiments, we use eight banks instead of two. Consequently, table entries are larger and occupy more area. Using the same area constraint, a predictor for eight banks must compromise on the number of entries per table and therefore suffers more collisions. In addition, the chance that a random prediction turns out to be correct is 50% for two banks compared to only 12.5% in our case. Zyuban and Kogge on the other hand use a synthetic predictor (a predictor that emulates a certain prediction rate and accuracy without implementing the internal structures of the predictor). A predictor with these properties would allow us to use these proposals in the design of our distributed memory unit. Unfortunately, we could not find a predictor that satisfies these requirements for a configuration of eight banks.

Sometimes it is possible to trade a lower prediction rate for a higher accuracy by generating more conservative confidence predictions. In this case, the predictor generates fewer confident predictions, but those predictions achieve a higher accuracy. We experimented with tables of saturating counters of varying bit–size, adopted more conservative approaches, where the counter is reset with every misprediction, and used various indexing schemes to access the table with the counters. While some of these schemes did slightly improve the accuracy, none of them was able to push it near 100%. Our inability to construct a bank predictor, which achieves an accuracy near 100%, finally lead us to consider different microarchitectural solutions than those described by Yoaz et al. and Zyuban and Kogge.

3.6 Energy and Performance Estimations

In Section 1.2, we did a simple estimation of the energy and performance of centralized versus distributed caches. The evaluation included one distributed configuration without a bank predictor and another with a perfect predictor. The results showed that a predictor can reduce the average access times as well as the dynamic energy consumption.

To conclude this chapter we show these results again and include an improved configuration, which contains a Tournament Predictor with tables of 2048 elements and a total size of 3Kbytes. For this estimation, we consider the predictors accuracy as well as its dynamic and static energy consumption:

$$E_{\text{Load}} = E_{\text{CacheRead}} + E_{\text{Communication}} + E_{\text{BankPredictor}}$$
$$E_{\text{BankPredictor}} = (1 + P_{\text{Unmatched}}) \cdot E_{\text{BankPredictorRead}} + E_{\text{BankPredictorWrite}}$$

To be consistent with the experiment in Section 1.2 we simulate a bank predictor for four banks. We calculate the energy usage of the predictor's memory using CACTI 6.5. [Mur09] For each cache access, we consider one predictor read and one predictor write. In addition, we account for unmatched load instructions, which are evicted from the pipeline by flushes before they can access the cache:

$$P_{\text{Unmatched}} = 0.2733$$
$$P_{\text{Hit}} = 1 - P_{\text{Miss}} = 0.7937$$

This probability as well as the bank predictor accuracy was obtained from simulations of our microarchitecture, which is described in detail in the following two chapters. The energy consumed by the bank predictor is offset by the reduced communication of load data. To calculate the average distance, which load data has to travel, we distinguish two cases: If the bank prediction is correct, no communication is required, if the bank prediction is incorrect, the load data has to travel 2 unit distances on average (compare Figure 1.1). Using the accuracy of the bank predictor, we can calculate the total average distance for load data:

$$d_{\text{Hit}} = 0$$
$$d_{\text{Miss}} = 2$$
$$d_{\text{LoadData}} = d_{\text{Miss}} \cdot P_{\text{Miss}} + d_{\text{Hit}} \cdot P_{\text{Hit}} = 0.4126$$

This average distance is well below the average distances for the centralized configuration (1.25 unit distances) and the naïve–distributed configuration (1.5 unit distances). We can therefore expect a considerable effect on latency as well as on dynamic energy consumption.

Figure 3.22 shows how the predictor affects access time and the dynamic energy consumption. The configuration with the Tournament Predictor is labeled as improved distribution. The access time is mainly determined by the accuracy of the predictor. For the majority of the load instructions, there was no inter–cluster communication required from the data cache to the destination register, just like in the case of the ideal distribution. Accordingly, the access times are close to the ideal distribution and from 17 to 38% lower than the centralized configuration, depending on the process technology.

Less inter–cluster communications also lead to less dynamic energy consumption. Unfortunately, the predictor itself consumes energy too, which offsets some of the savings.

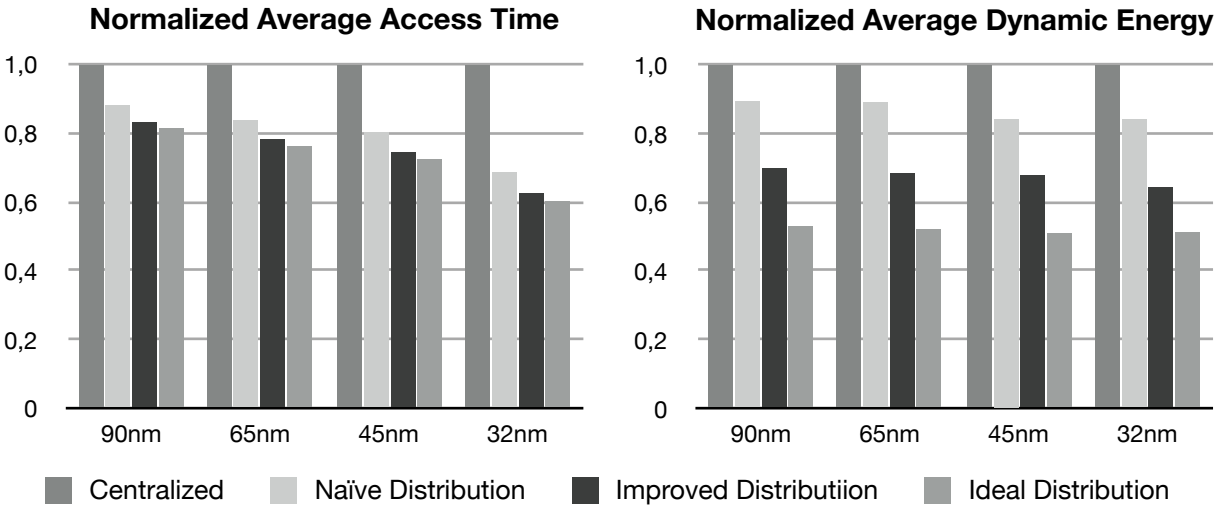

Figure 3.22: Revisiting Access Time and Dynamic Energy. The dark bar shows the estimation for an improved distributed configuration with a bank predictor (the Tournament Predictor). This figure is based on Figure 1.2 in Chapter 1.

Nonetheless, the result is still a significantly reduced energy consumption. When we compare the dynamic energy of the improved to the ideal distribution, about half of the difference is due to communications as a consequence of bank mispredictions and the other half is due to the consumption of the predictor itself. Compared to the centralized configuration dynamic energy is 30 to 36% lower, depending on the process technology.

The predictor also increases the static energy consumption. Including cache, interconnects, and the predictor, the real distribution consumes only slightly more static energy than the centralized configuration, depending on the process technology from 0 to 2%.

We conclude that using bank predictors provides important benefits for a configuration with distributed caches and makes it much more attractive than a centralized cache configuration. This evaluation ignored an important step in the memory pipeline of an out-of-order processor: the disambiguation of memory dependencies. We will discuss this topic in depth in the next chapter.

3.7 Conclusions

In this chapter, we adapted predictors from other domains to the problem of bank prediction and quantitatively evaluated the resulting predictors for a configuration of eight banks. Up to a size of approximately 3Kbytes, the bank predictor based on the Gshare branch predictor delivers the best accuracy. For greater sizes, a Tournament Predictor is the best choice. We found that even though bank predictors can deliver an accuracy of up to 96% when the predictor size is unconstrained, realistic configurations of about 3.3Kbytes reach an accuracy of 84%. Including a confidence for each prediction was not sufficient to reach an accuracy near 100% for realistic predictor sizes. This implies that our microarchitecture proposal requires an efficient way to handle bank mispredictions.

CHAPTER 4

A DISTRIBUTED MEMORY UNIT

Baseline Clustered Architecture

The clustered architecture, which we use as a basis for our design and experiments, is described in detail by Parcerisa [Par04]. The superscalar architecture is divided in a centralized front-end and a distributed back-end. The front-end consists of the stages Fetch, Decode, Cluster Steering, Rename, and Dispatch, the back-end of the stages Issue, Register Read, Execute, Write–back, and Commit. Structures that belong to the back-end such as issue queues, register file, and functional units are all distributed over homogeneous clusters. A cluster therefore contains an issue queue, a register file, and various functional units. Instruction results inside a cluster are handled (e.g. forwarded) like in a conventional superscalar microarchitecture. If a result is needed at another cluster, a special copy instruction is inserted in the instruction stream during the rename stage. This copy instruction copies the data over an interconnection network to another cluster. Instructions are assigned to clusters during the Cluster Steering stage. For best performance the steering algorithm attempts to balance two contradicting goals: to minimize inter–cluster communication and to make the best use of the resources of all clusters.

The data memory pipeline of the base architecture is centralized and not distributed. This gives rise to the following design proposals for a distributed memory unit.

4.1 Qualitative Design Decisions

4.1.2 Distributed Cache

The baseline architecture assumes a single memory pipeline and a centralized data cache. There are two motives to adopt distributed memory pipelines and data caches and associate a memory pipeline and a cache bank to each cluster.

First, the memory pipeline contains some very expensive logic, most notably the content associative memory of the memory queues used for disambiguation. This pipeline can be sliced in multiple simpler pipelines as described by Yoaz et al. [Yoa99] The key idea is to use part of the data address to assign instructions to the pipeline in the same way as data is assigned to banks in a banked cache. Using this way of distribution guarantees that memory dependencies are confined to a single cluster and inter–cluster memory dependencies are avoided. This greatly simplifies the disambiguation logic and the content associative memory of each pipeline can be significantly smaller than the memory of a single centralized pipeline.

Second, the distributed data caches allow for the reduction of the physical distance between data cache, memory pipelines and functional units. This results in better access times and lower power consumption. While it is hard to keep all memory accesses local so that addresses and data travel only inside a single cluster, even a partial reduction of inter–cluster communication improves performance and lowers power consumption. See Section 1.2.2 in Chapter 1 for a quantitative evaluation.

Another alternative to the centralized and distributed cache is a replicated cache where each cluster holds a copy of the whole data cache. This way all read accesses to the data cache are local to a cluster while write accesses must be broadcast to all other replicas of the data cache. This alternative has the obvious disadvantage that the effective cache size is reduced. But the most serious disadvantage for our application is that it does not allow a straightforward distribution of the memory pipelines like a distributed banked data cache does. In the following, we will therefore concentrate on the option of a distributed cache.

4.1.3 Inter–Cluster Networks and Instruction Steering

Any given load instruction may travel up to three clusters in its lifetime. Our baseline architecture is based on the Digital Alpha ISA and therefore the address calculation of a load instruction can have at most one register operand. Therefore, the first cluster visited is the cluster where the source register is located. After the address calculation, the load travels to the cluster where its data is located. It enters the memory pipeline, performs disambiguation, and obtains its data from the cache. Finally, the instruction has to travel to the cluster where its destination operand was mapped to write its data into the register file and forward it to all instructions waiting for the data.

The problem is illustrated in Figure 4.1. The first trip of the load instruction from cluster A to cluster B could be avoided if the source register and the load data were mapped to the same cluster. Unfortunately, there is no easy way to achieve this because the mapping of the load data depends entirely on the address, and the mapping of the register operand may occur a long time before the load instruction is even fetched. Instead, we chose to provide a dedicated inter–cluster network to transport load instructions from the functional unit where the address is calculated to the memory pipeline where the instruction is disambiguated.

The second trip of the load instruction from cluster B to cluster C can be avoided if the output register is mapped to the same cluster as the load data. Unfortunately, the cluster of the output register is scheduled during the Cluster Steering stage when the actual address and the cache bank of the load data is still unknown. However, we can use a bank predictor like the one described by Yoaz et al. to predict the cache bank and thereby the cluster where the load data is located. If we map the output register to the predicted cluster the second trip from B to C becomes unnecessary for the majority of load instructions. However, we have seen in the last chapter that it is not reasonable to expect a bank prediction accuracy near 100% for all load instructions. Therefore, we still need a way to gracefully handle bank mispredictions. If the cache bank was predicted incorrectly and the output register was mapped to another cluster, we transport the load data to its destination over a second dedicated inter–cluster network.

Our proposed solution is different from Yoaz' et al. solution in that they explicitly abstain from adding any communication link between memory pipelines. For Yoaz et al. this approach is feasible because they limit themselves to a microarchitecture with a shared register file and only two memory banks for which high bank prediction rates and an accuracy near 100% are realistic.

Zyuban and Kogge [Zyu00][Zyu01] also add two inter–cluster networks for addresses and data respectively to their distributed memory unit. In their scheme, these networks are used by low–confidence loads and by mispredicted load instructions. For high–confidence load instructions, the address calculation is steered to the cluster where the load data is predicted to be located and the source operand is sent to this cluster using the operand inter–cluster network. If the prediction turns out correct no further inter–cluster communication is necessary but if the prediction is wrong, two more inter–cluster communications are needed to obtain and to return the load data. Therefore, in the worst case of a mispredicted high–confidence load instruction up to *three* inter–cluster communications are necessary, each on a different inter–cluster network. Because a combination of high prediction rate with a near 100% accuracy is very difficult to obtain for more than two clusters Zyuban and Kogge's scheme requires notably more inter–cluster communications than our proposal.

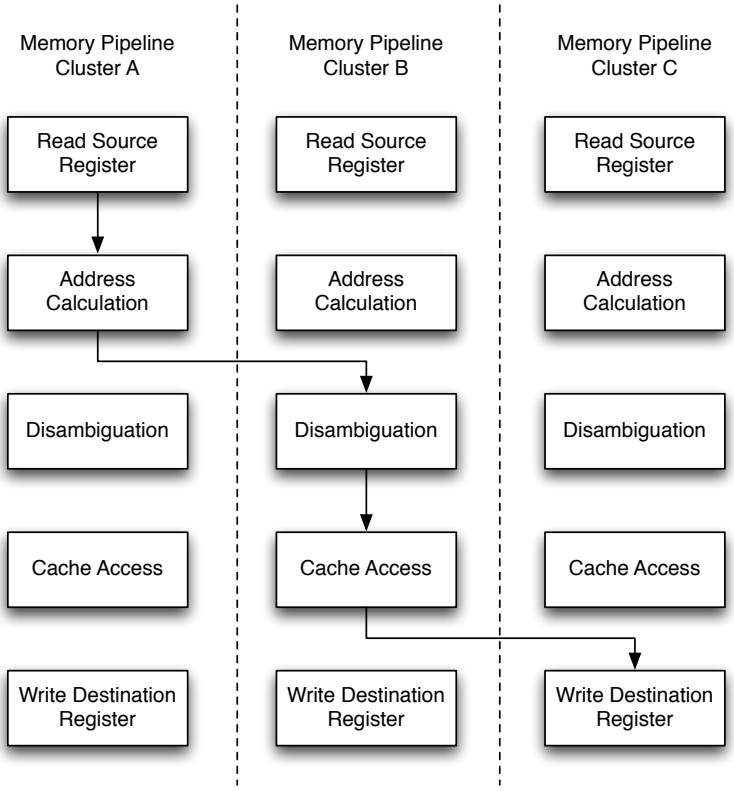

Figure 4.1: Example of a load instruction that travels three clusters.

4.1.4 Late allocation and unordered queues

An important consideration in the design of a distributed memory unit is the amount of time during which instructions occupy entries in the memory queues. Traditionally, these entries are assigned to memory instructions before they are dispatched to the issue queues. Thereby this assignation proceeds in the original program order. As soon as the address is calculated and the instruction enters the memory pipeline, its queue entry is actually occupied and filled with data. The entry then remains occupied until the instruction commits.

Late Allocation

Reserving entries in the various queues of a distributed memory unit is much more difficult because during the Dispatch stage the address is unknown, and it is therefore not clear in which of the multiple queues an entry should be reserved. Yoaz et al. propose to use bank prediction to reserve entries in the instruction queues. They reserve one entry for each high–confidence prediction and entries in all queues for low–confidence predictions. Once the address of a low–confidence instruction is known, the unused entries are cancelled. Cancelled entries are still unavailable for immediately following instructions unless an (overly) expensive compacting queue design is used. To avoid excessive canceling of memory entries the prediction rate (the percentage of high–confidence predictions) must be high. On the other hand, the accuracy (the percentage of correct high–confidence predictions) must be near

100% because it is very difficult to recover from mispredictions. In such a case, an entry would be reserved in the wrong queue and had to be moved to the right queue while maintaining all queue entries in the original program order. This would require an expensive queue design that allows the insertion of entries or (more likely) involve an expensive pipeline flush. Because our experiments on bank predictors described in the last chapter show that it is very difficult to obtain both a high prediction rate and near 100% accuracy at the same time we discarded the possibility of adopting Yoaz et al. scheme for our distributed memory unit.

Unordered Queues

In the description so far, we assumed a traditional disambiguation logic that requires that entries are physically ordered in the original program order. Therefore, the entries have to be assigned to instructions at an early pipeline stage before the address (and hence the right memory queue) is known. If we could assign the instructions to queue entries in any order, the assignment could be delayed until the addresses were actually known. These lately allocated and unordered queues would be better utilized too because the memory instructions would spend less of their lifetime in the queue. Therefore, the queue size could be reduced and some power could be saved without affecting performance negatively. However, the main advantage is that unordered queues do not depend on unrealistically accurate bank predictors to work efficiently, because mispredicted instructions may enter the queue out of program order and without requiring a pipeline restart.

Separate Load and Store Queues

Our proposal uses the Alpha instruction set architecture and we adopt some of the peculiarities of the Alpha EV6 like the separate queues for load and store instructions [Kes98]. However, we depart from it in other details. E.g. we split store instructions into internal store address and store data instructions. In many cases, the store address is known significantly earlier than the store data and can be used by the disambiguation logic to resolve potential memory conflicts. (See Section 4.1.6 below for more details.)

4.1.5 Deadlocks

Unfortunately, allocating resources such as queue entries out of program order can have undesirable effects. One of them is the risk to create deadlocks. Deadlocks can occur when the calculation of an address is delayed so long that all queue entries in the right cluster are already occupied by younger instructions. Because queue entries are usually deallocated during the Commit stage and therefore in program order none of the younger memory instructions will ever deallocate its entry before the waiting instruction. Since a full memory queue is a precondition for a deadlock, small queues are more prone to deadlocks than large queues. Recovering from a deadlock is not trivial and we use a pipeline flush to be able to continue processing.

We will discuss more sophisticated solutions to handle and prevent deadlocks in the next chapter. (See Section 5.1 and Section 5.4 in Chapter 5.)

4.1.6 Reservation of Queue Entries

In a traditional organization with ordered queues, the reservation of queue entries is straightforward. A simple counter is sufficient to assign queue entries to instructions. An unordered queue requires a different approach. The problem is very similar to the allocation of physical registers and the corresponding solutions can be adopted for the reservation of load queue entries. For small queues a bit–vector will require fewer resources than a free list. However, for larger queues where multiple entries have to be allocated and released in each cycle a solution applying free lists may be more appropriate.

4.1.7 Release of Queue Entries

When an instruction is committed and subsequently removed from the reorder buffer the corresponding cluster and the queue entry number must first be identified before the queue entry can be released.

For this purpose, each cluster of the distributed memory unit contains a table for each local queue that maps instructions to local queue entries. This table uses the global memory sequence numbers of the load and store instructions as index and contains the number of the local queue entry. The global memory sequence number is assigned to each instruction in the front-end and is used in various places inside the microarchitecture to establish the relative age of memory instructions. (Load and store instructions are numbered separately.) Whenever an instruction is inserted into a local queue, its table entry is updated accordingly.

When a memory instruction commits, the ROB broadcasts a message to all clusters of the distributed memory unit. This message specifies how many load and how many store instructions are to be committed. Apart from these two numbers, no more information needs to be sent. The clusters each contain a commit pointer into their tables and therefore have sufficient information to liberate local queue entries as required. This implementation is reasonably simple and allows to reserve and release several queue entries per cycle.

Section 5.5 in the next chapter describes an extension to this release mechanism.

4.1.8 Handling Store Data and Store Address Instructions

Store data and store address instructions must be associated in the memory unit to provide the proper function of the memory system. In a traditional microarchitecture with a single memory queue this is easy to achieve because the allocation of queue entries occurs when the instruction stream is still in the original program order and store data and store address instructions are not yet separated.

The first problem is to guarantee that store data and store address instructions are routed to the same cluster of the memory unit. The store address instruction is routed using the result of its address calculation. The store data instruction should arrive at the same memory pipeline as the store address instruction. To achieve this we make use of the fact the microarchitecture supports one register output and up to two register inputs. This allows us to use a register to pass the cluster number between the two instructions; i.e. the store address instruction writes the cluster number to a register where the store data instruction picks it up and follows its sibling to the same memory pipeline.

After the address has been calculated, the store address instruction is the first of the two sibling instructions that arrives at the destination cluster. Upon arrival, it is inserted into the store queue as described above. When the store data instruction arrives, it must identify the store queue entry of its sibling. To associate both instructions we use the same table described above in Section 4.1.7 that maps global sequence numbers to store queue entries. Once the corresponding entry in the store queue is identified, the store data is written into the store queue entry.

4.1.9 Store–to–Load Forwarding

To enable the forwarding of data from a store to a load instruction, load instructions search the store queue when they are executed. If no match is detected the load obtains its data from the memory hierarchy (the data cache in most cases). If a single match is detected the age of the load and store instructions is compared. Store–to–load forwarding is enabled only if the load is younger than the store (i.e. if the load follows the store in the original instruction stream). To compare the instruction age the sequence numbers of the two instructions are subtracted. For the microarchitectural parameters, which we use in our simulations, these sequence numbers are nine bits long. Consequently, an age comparison requires only a single, nine–bit–wide subtraction.

If more than a single match is detected the forwarding logic must choose one of the matches for forwarding. We handle this case using no–hit bits as described in Sections 2.2.4 and 2.2.5. For this scheme a no–hit bit is added to each queue entry. Whenever a store instruction is inserted into the queue, its address is compared to the addresses of store instructions already present in the queue. If a store instruction in the queue matches, the no–hit bit of the older of the two instructions is set. At any point in time, out of several stores in the queue with matching addresses, all but one will have the no–hit bit set. Instructions with the no–hit bit set are exempt from all future address comparisons. As a result, load instructions that search the store queue never match more than a single store instruction.

There is one downside to this technique: if the load instruction hits a store instruction that is younger (i.e. a store that follows the load in the original instruction stream) there may be older store instructions present in the queue that are not detected because their no–hit bit is set. Since this is an infrequent event it is acceptable to simply flush the pipeline to recover.

This condition is detected by comparing the sequence numbers of the load instruction and the matching store instruction.

In addition to memory address and instruction age, the forwarding logic must take the access width into account. The Digital Alpha ISA allows accesses of one, two, four, and eight bytes width. All accesses are aligned, i.e. the memory address is always a multiple of the access width. The disambiguation logic described above operates at a granularity of 64–bit words, and the alignment rules of the ISA guarantee that all accesses are fully contained in a single 64–bit word. Each load and store queue entry contains a bit-mask, which marks the corresponding bytes inside the 64–word. This bit-mask allows a quick check, if all the required bytes for the Store–to–Load forwarding are contained in a single store queue entry. A shifter inside the forwarding logic aligns the data from the store queue for the load instruction. Load instructions, which cannot obtain their data from the memory hierarchy or from the Store–to–Load forwarding logic, are delayed until the offending store instructions commit. In the absence of a structure to hold delayed load instructions (see Section 5.3 in the next chapter), this delay causes a pipeline flush.

4.1.10 Handling Unresolved Stores

In the scheme described above, load instructions search only for store instructions present at execution time in the store queue. Store instructions only take part in the disambiguation after their address is known. The disambiguation of load instructions is therefore speculative. To check if any dependencies were violated by this speculative disambiguation store instructions search the load queue for younger loads (i.e. loads, which follow the store in the original instruction stream) with matching address as soon as they are inserted into the store queue. If such a load is identified the microarchitecture recovers with a pipeline flush. We provide only one comparator circuit for this test and use the no–hit bit for load queue entries in a similar manner as for the store queue described above, i.e. for all the load instructions that match the address the no–hit bit is used to suppress all but the youngest hit.

To reduce the number of memory dependency violations we include a store wait table similar to the Alpha EV6 in our microarchitecture, see Section 2.2.4 and Kessler et al. [Kes98] Load instructions, which are predicted to violate dependencies, are retained in the issue queue before the address calculation until the addresses of all older store instructions are known. In Section 5.3 of the following chapter, we will introduce the *memory issue queue,* a structure, which can hold these delayed load instructions. Using this queue, we will be able to avoid wasting expensive issue queue entries. Section 4.2.3 below will give a rationale for this choice of load issue policy.

4.1.11 Multiprocessor Memory Consistency

The requirements for multiprocessor memory consistency vary with the instruction set architecture in question. Since we adopt the Alpha instruction set architecture for our experiments, we also adopt its memory consistency model.

The consistency model requires that we detect cases where two speculative load instructions access the same memory location in reverse order, where the younger load (i.e. the load that follows the other in the original instruction stream) accesses the location before the other load does.

To test for this condition each load instruction searches the load queue for instructions with the same address. As mentioned previously this search and the following age comparison are required to establish the no–hit bits in the load queue. This scheme is modified to detect the case where the load that is about to be inserted into the queue hits a load instruction in the queue that has the same address but is younger (i.e. the load in the queue follows the other load in the original instruction stream). When this case is detected, the pipeline is flushed to avoid a violation of the memory consistency model.

4.1.12 Recovering From a Pipeline Flush

Sometimes it is necessary to flush the pipeline to recover from an error condition. The most frequent of these conditions is a failed branch prediction. Some of these events are part of the instruction set architecture, like external interrupts, floating-point exceptions, memory access exceptions etc. Some infrequent conditions are handled with flushes to avoid the otherwise required implantation complexity, some examples are described above in Sections 4.1.5, 4.1.9, 4.1.10 and 4.1.11.

In every pipeline flush, some offending instruction is identified. This instruction and all instructions following it must be removed from the pipeline while all instructions preceding it should continue execution in the usual manner. To avoid a complex and slow mechanism that removes instructions from the memory queues on an individual basis the distributed memory unit removes all instructions at once. This operation is akin to a reset—after it is performed all queues are empty. To speed up the process of a pipeline flush the front-end and the back-end are not flushed at the same time. The front-end is flushed immediately after the pipeline flush is triggered and then starts operation by fetching the offending instruction again from memory. The back-end continues normal operation until the offending instruction reaches the commit stage, but instead of committing the instruction, the back-end is flushed. To guarantee correct execution no instructions are allowed to pass from the front-end to the back-end during the time between the flush of the front-end and the flush of the back-end. Flushing the front-end in advance of the back-end helps to speed up the recovery. Other structures of the back-end such as issue queues also use the same mechanism. Papworth et al. describe this technique in more detail. [Pap96]

4.2 Quantitative Design Decisions

In this section, we evaluate design options in a quantitative manner.

4.2.1 Experimental Methodology

To evaluate our microarchitectural proposals we use a simulator, which is based on the work of Parcerisa, [Par04] which in turn is based on the SimpleScalar toolset. [Aus02] The simulator uses the Digital Alpha instruction set architecture. We use the SPEC CPU2000 benchmarks to evaluate our architecture proposals. The benchmarks are compiled and optimized for the Alpha EV6 using the original toolchain from Digital. We simulate the first 100 million instructions after skipping the initialization phase of each benchmark[14]. Table 4.1 summarizes the most important architectural parameters. This methodology is used for chapters 4 and 5 of this thesis.[15]

Frontend (Monolothic, In-Order)	
WidthBrei32	8 instructions
Depth	9 stages
Branch Predictor	tournament: bimodal+gshare, 6Kbytes
Branch Target Buffer	64K entries, 4ways
Return Address Stack	64 entries
Reorder Buffer	256 entries
Backend (Clustered, Out-of-Order)	
Number of clusters	4 clusters
ALU (per cluster)	
units / registers / issue queue size	2 units / 56 registers / 16 entries
FPU (per cluster)	
units / registers / issue queue size	1 unit / 56 registers / 16 entries
Interconnection Network	
topology / latency per hop	asynchronous ring / 1 cycle
Memory Hierarchy	
Level 1 data cache (per cluster)	
size / associativity / line size / latency	16Kbytes / 2 ways / 32 bytes / 2 cycles
read / write ports	1 port / 1 port
Level 1 instruction cache (monolithic)	
size / associativity / line size / latency	64Kbytes / 2 ways / 32 bytes / 1 cycle
Level 2 unified cache (monolithic)	
size / associativity / line size / latency	2Mbytes / 16 ways / 32 bytes / 14 cycles
Main Memory	
random / sequential access latency	96 cycles / 13 cycles
Bank Predictor	tournament: gshare+local, 3.3 Kbytes

Table 4.1: Main architectural parameters of the microarchitecture.

[14] The initialization phase of each benchmark was determined by inspecting and instrumenting the source code of each benchmark.

[15] With the exception of Section 4.2.2 of chapter 4, which uses a more aggressive configuration.

4.2.2 Choosing the Interleaving Factor

The interleaving factor decides which bits of the address indicate the cache bank. The interleaving factor defines the mapping of memory locations to cache banks and therefore to clusters. We consider factors from eight to 512, smaller factors than eight are not reasonable because we are using a 64–bit architecture and a smaller factor would spread a single memory word over several banks. Another important number in this context is the cache line size. If the cache line size is greater than the interleaving factor, a single cache line is distributed over several banks. In this case several clusters will require copies of the same tag to detect cache hits and misses in an autonomous manner without requiring inter–cluster communications. The extreme case is an interleaving factor of eight, where every 64–bit memory word possesses a copy of the tag. If a cache line is distributed over several clusters, the mechanisms for replacing cache lines are significantly more complicated.

Two more observations influence the choice of the interleaving factor. The interleaving factor defines the mapping of memory locations to clusters. This is of importance to the bank predictor and impacts the prediction accuracy as illustrated below in Figure 4.2. Finally, the interleaving factor affects the number of bank conflicts when these banks have a limited number of ports. In general, smaller factors lead to fewer conflicts as illustrated above in Figure 4.3.[16]

Figure 4.2 shows the effect of the interleaving factor on the accuracy of a Tournament Bank Predictor of 3.3 Kbytes on some selected benchmarks of SpecINT. The arithmetic mean is shown on the right side of the figure. Each benchmark shows its own characteristic. While e.g. *bzip2* is mostly unaffected by the choice of the interleaving factor, others like *perl* show significant differences. In general, higher factors lead to a higher accuracy. However, the lowest overall accuracy is not produced by the smallest interleaving factors—as might be expected—but by factor 32 (bits 5:7).

Figure 4.3 shows the effect of the interleaving factor on the relative IPC for a configuration with a single read and a single write port per bank relative to a configuration with unlimited ports. The harmonic mean of the benchmarks is shown on the right of the figure. For this experiment, we used configurations with 8 clusters and a Tournament Bank Predictor. The results show a steady decrease of performance between 1% and 0.5% for each bit of the interleaving factor. This decrease in performance is caused by the increased number of bank conflicts and an increasingly unbalanced distribution of the data.

Considering especially our earlier points on an easy implementation we choose an interleaving factor that is equal to the cache line size of our architecture, which is 32 bytes.

---

[16] Results shown in Figures 4.2 and 4.3 were obtained with a more aggressive configuration than the one presented in Section 4.2.1. Instead of 4 this configuration employs 8 clusters and 8 cache banks.

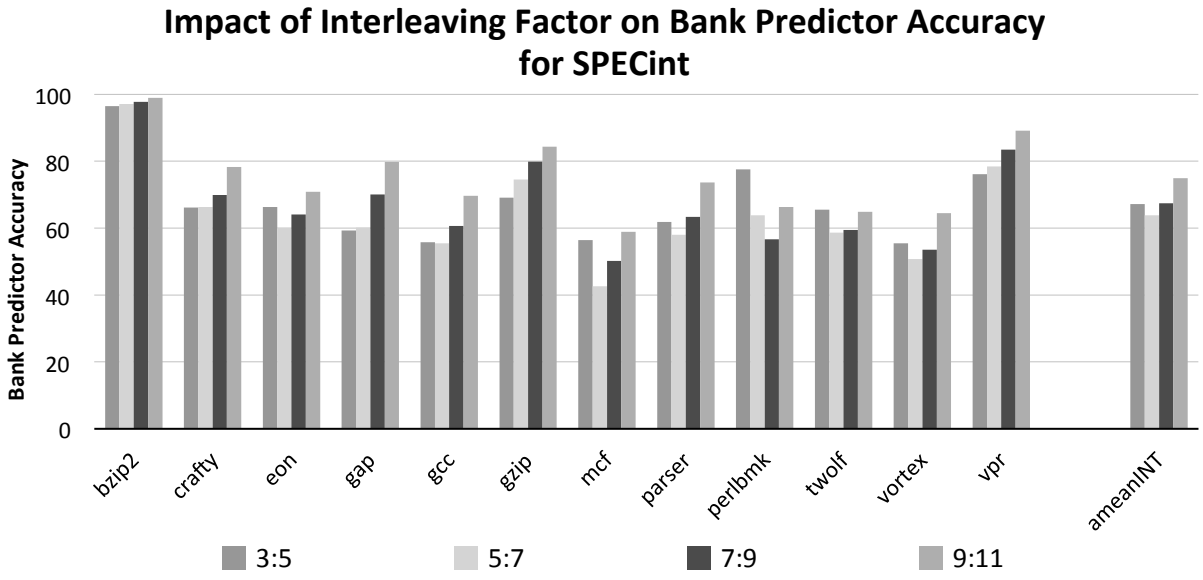

Figure 4.2: The effect of the interleaving factor on accuracy. We employ a Tournament Bank Predictor of 3.3 Kbytes.

Figure 4.3 also shows that a single read port is sufficient for each cluster (there is a penalty of only 2%) therefore we provide each cluster with a single read and a single write port.

4.2.3 Load Issue Policy

The architecture described so far disambiguates instructions speculatively, store instructions, which are not present in the store queue (i.e. stores with unresolved addresses), are ignored by loads, even if the store is older and a dependency might exist, see Section 4.1.10 above. If many dependencies are violated and the pipeline has to be flushed often this speculation can have an overall negative effect on performance. Therefore, we compare in this section four different load issue policies:

- a conservative policy where loads are only allowed to execute if all older stores are present in the queue,
- a speculative policy that ignores stores that are not present in the queue,
- a speculative policy with a predictor, a store wait table like the one described by Kessler et al. [Kes98], and
- a speculative policy with an oracle. The oracle (a perfect predictor) uses Cain and Lipasti's value based definition of memory dependencies.[17] [Cai04] This configuration serves as a baseline to normalize results.

[17] By this definition a memory dependency is violated only if a load instruction returns an incorrect value. In contrast to the classic definition the value based definition allows silent stores (stores, which do not modify memory contents) to access memory without causing a dependency violation.

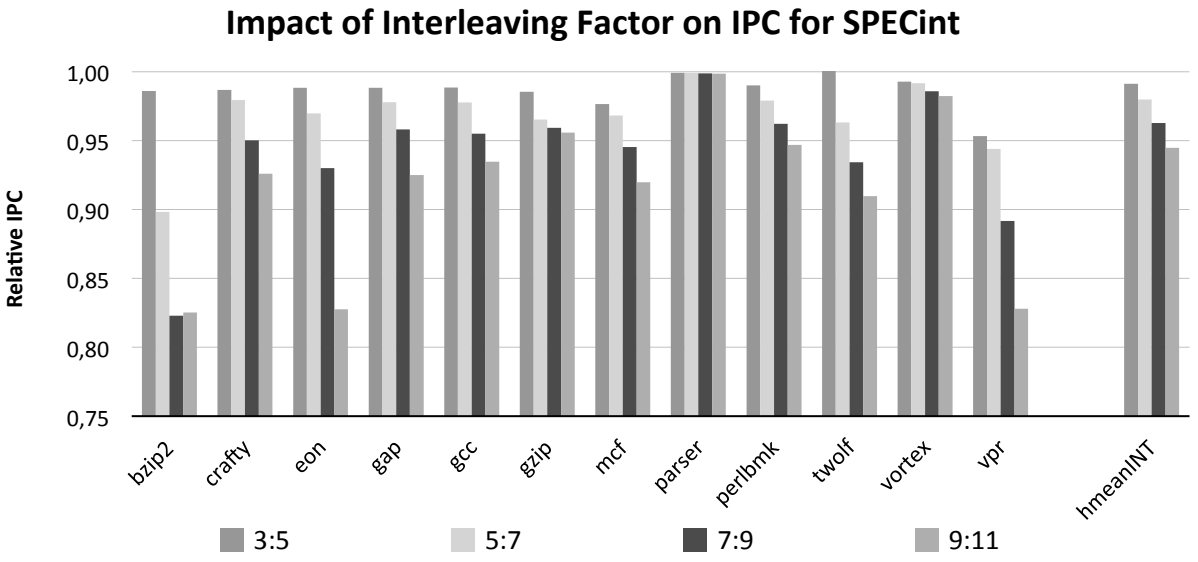

Figure 4.3: The effect of the interleaving factor on performance. The performance shown is for a configuration with only one read and one write port per cluster relative to a configuration with infinite read and write ports.

Figure 4.4 and Figure 4.5 show the results of the experiment. The harmonic mean can be seen on the right side of the figures.

All configurations include a memory-issue queue, which will be presented in Section 5.3 of the next chapter. This queue can buffer load instructions, which have to wait for the completion of older stores. Without the memory issue queue these load instructions would have to wait e.g. in the integer issue queue and would thereby reduce the effective window size of the out-of-order core. We include the memory issue queue in this experiment to avoid measuring this type of secondary effects.

The most striking result in above figures is the low performance of the conservative disambiguation scheme. One of the causes for the low performance is the delay of the additional communications between clusters, which are required to implement the semantics of conservative memory disambiguation. Before a conservative disambiguation mechanism can allow a load instruction to execute, it needs to know the status of all store instructions that precede the load in program order. The load can be allowed to execute only if all previous stores either execute in other memory pipelines or can be disambiguated locally using their address. This requires a broadcast of the status of all store instructions to all memory pipelines and thereby introduces an additional delay. This delay affects the conservative disambiguation scheme most.

Another motive for the low performance of the conservative scheme may be the fact that the binaries for our simulations have been optimized for an Alpha EV6, which uses speculative load issuing with a predictor similar to the third configuration in our experiment. This could benefit the configurations, which use speculative load issue. Finally, our

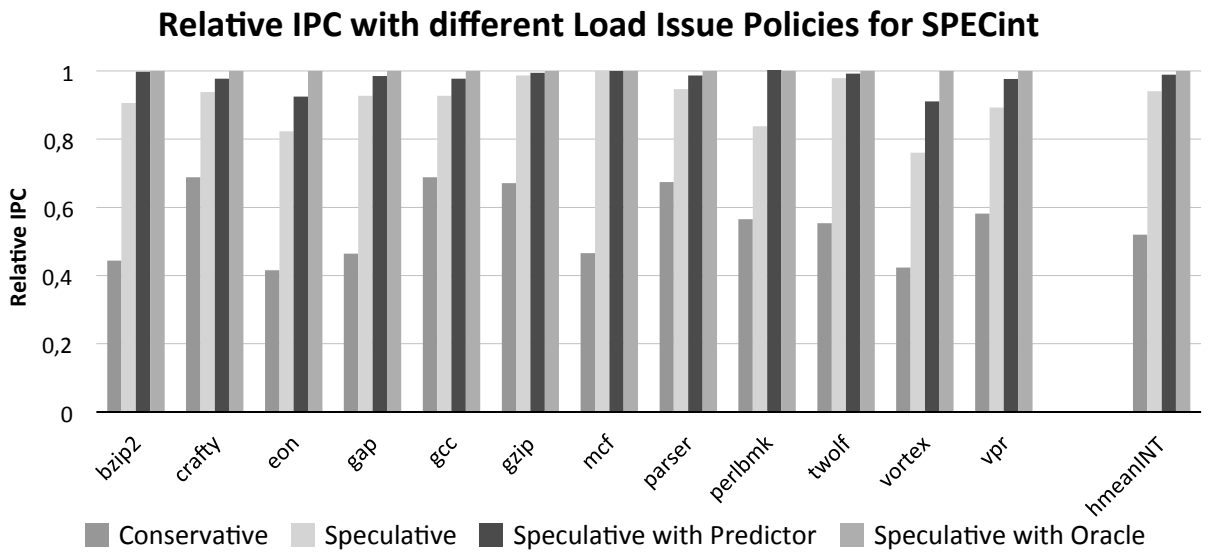

Figure 4.4: Relative IPC with four different load issue policies for SPECint. The configuration using speculative disambiguation with an oracle serves as baseline.

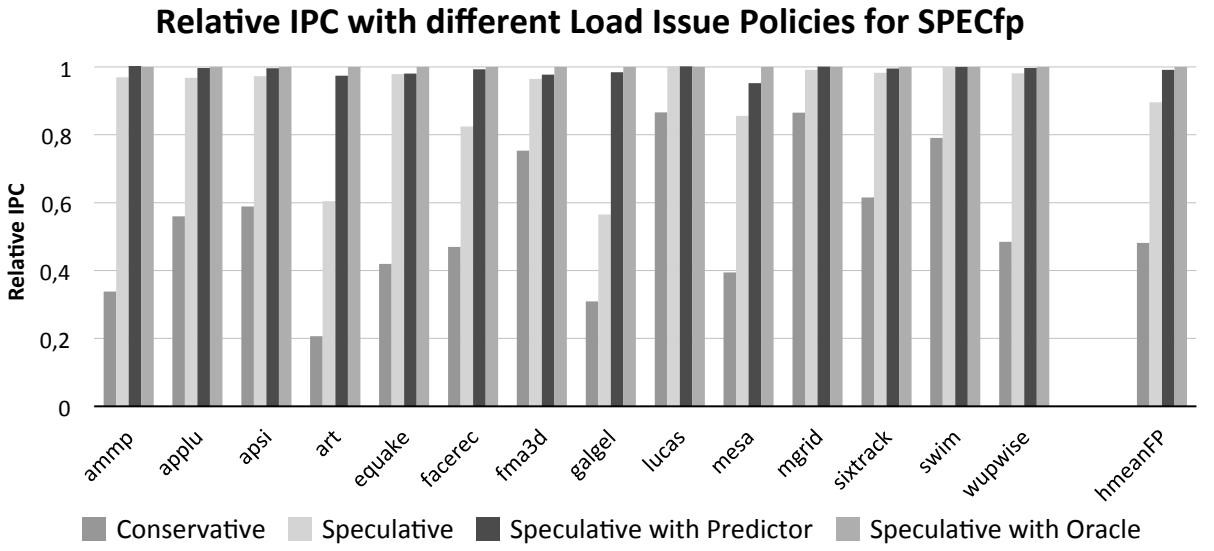

Figure 4.5: Relative IPC with four different load issue policies for SPECfp. The configuration using speculative disambiguation with an oracle serves as baseline.

architecture splits store instructions in store address and store data instructions. This allows store address instructions to participate in the disambiguation even though the respective store data is not yet known. This is of advantage to conservative as well as speculative schemes.

The results also show that a dependency predictor using a store wait table is very effective. A relatively simple predictor can achieve a performance, which is very close to the ideal baseline (within 1% on average) and consistently better than the configuration without a predictor.

Given above results we include the speculative load issue policy using a store wait table dependency predictor in our design proposal.

4.3 Conclusions

In this chapter, we presented the design of a distributed memory unit. This memory unit employs a reservation scheme for the memory queues, which unlike previous proposals does not depend on a bank predictor. Memory queue entries are reserved just before instructions issue, which improves the utilization of the queues. As a consequence of the allocation being late, the allocation is performed out of program order, and the queues are unordered. We show that such an unordered queue may be implemented with known techniques at a cost comparable to a traditional age ordered queue.

We outlined the rationale behind our choice of the interleaving factor and showed how it affects bank predictor and performance. We show that a speculative load issue policy is essential for performance and that a simple dependency predictor delivers almost the same performance as a perfect predictor.

We outlined our design decisions and the rationale behind our choices. The memory unit described in this chapter will serve as a starting point and as a baseline for the proposals of the next chapter.

CHAPTER 5

IMPROVEMENTS TO THE DISTRIBUTED MEMORY UNIT

In this chapter, we present techniques to improve the distributed memory unit we presented in the last chapter.

The distributed memory unit we presented in the last chapter does not attempt to control the flow of memory instructions in the memory pipeline. This lack of flow control leads to load and store queue overflows and deadlocks that cause pipeline flushes, which reduce performance and energy efficiency. Memory instructions are mapped to one of the distributed queues based on the memory address. Because the address is calculated just before an instruction is inserted into a memory queue, the mapping is established very late and out of program order. In comparison, architectures without distributed queues can map memory instructions to queue entries at early pipeline stages and in program order. The distributed memory unit on the other hand requires a novel approach to control the flow of instructions.

The dispatch stage is the last pipeline stage of the front-end where instructions are processed in program order. This property makes it attractive to flow control. In later out-of-order stages, stalling instructions is more complicated. In Section 5.1, we will describe how this technique can be used to reduce or even completely eliminate overflows and deadlocks of the memory queues.

Since the essential cause of queue overflows is a lack of free entries, an obvious solution is to enlarge the queues. While this solution does work, it is also very expensive. Load and store queues contain content associative memory, which is both slow and energy hungry. So instead of extending the load and store queues, we use a simple FIFO buffer to hold instructions until the instruction can be added to the main queue. Because instructions are simultaneously added to the queues and access the cache, we call this buffer the *pre–access queue*. This queue is described in Section 5.2.

The fundamental problem with the pre–access queue is that only the instruction at the head of the FIFO can issue to memory. It does not allow to choose the next instruction to issue. This prevents us from prioritizing instructions or from employing an issue policy. E.g. performance benefits from prioritizing instructions by age and the load issue policies presented in Section 4.2.3 of the last chapter have a mayor impact on performance. To resolve this problem we substitute the pre–access queue with a *memory issue queue*. This queue is described in Section 5.3.

The pre–access queue and the memory issue queue can avoid overflows of the memory queues but do not resolve the problem of deadlocks. Deadlocks can occur if an older instruction finds a full memory queue filled with younger instructions. A very conservative flow control mechanism in the dispatch stage can avoid deadlocks but seriously affects performance. The memory issue queue allows us to implement a load issue policy, which avoids deadlocks without compromising performance. This scheme is called the *conservative deadlock aware entry allocation* and is presented in Section 5.4.

Finally, we present a mechanism, which allows load queue entires to be released before the commit stage under certain circumstances. The *early release of load queue entries* reduces the pressure on the load queue. The mechanism is described in Section 5.5.

5.1 Dispatch Throttling

The design of the distributed memory unit presented in the previous chapter suffered from a large number of pipeline flushes caused by deadlocks and overflows of the memory queues. (See Figure 5.1)

By limiting the flow of memory instructions in the dispatch stage, a mechanism can effectively control the number of memory instructions in flight in the out-of-order backend. Supposing for a moment a design with a single centralized memory queue, if the number of memory instructions in flight is kept below or equal to the size of the memory queue, overflows can no longer occur. Deadlocks would also be avoided, since a full memory queue is a prerequisite for a deadlock.

Unfortunately this idea cannot be applied directly to the proposed distributed design because it includes multiple address–interleaved memory queues instead of a single

centralized memory queue, and the addresses—and therefore the mappings of instructions to queues—are not yet known in the dispatch stage.

The throttling mechanism can however take into account all those instructions that did already calculate their address. Problematic are instructions with unknown addresses. There are several ways to handle these instructions. In the following, we will examine three strategies. The *conservative* strategy is to plan for the worst case in which all instructions with unresolved addresses would go the queue with the least free space left. A more *optimistic* approach uses the bank predictor to stall the dispatch stage when the predicted mapping indicates an overflow. Finally, a *hybrid* method combines the two previous techniques using the confidence of the bank predictor. If the predictor is confident that its prediction is correct, the optimistic approach is used, otherwise the conservative method is applied.

Naturally, the conservative strategy avoids overflows and deadlocks completely, while the number of overflows and deadlocks in the other two schemes depends on the accuracy of the bank predictor. Consequently, the number of flushes will be less with the conservative strategy. However, the conservative strategy stalls the dispatch stage more often than the other two schemes do. This tradeoff between flushes and stalls is present in all three proposed strategies and is fundamental to the dispatch–throttling scheme.

When a conservative dispatch throttling is used together with the pre–access queues, the scheme cannot cause overflows but it can still cause deadlocks. Deadlocks can occur, because the dispatch throttling allows both the memory queue *and* the pre–access queue to fill up completely, but does not take the relative distribution of instructions between the two queues into account. If the memory queue is full and the oldest instruction is not located in the memory queue, no further progress can be made and a deadlock occurs (see Section 4.1.5 of the previous chapter).

5.1.1 Dispatch Throttling: Implementation

Dispatch throttling (or flow control at dispatch, FCD for short) can be implemented relatively straightforward with a set of counters. Each counter corresponds to a memory queue and determines how many instructions are in flight in the backend that are mapped to this memory queue. E.g. a design would contain one counter for the load queue and another for the store queue, with 4 clusters there would be 8 counters in total. If any of the counters is greater than the queue size, the dispatch stage stalls.

The conservative technique would increment all load queue counters by one for each load instruction that passes dispatch. Similarly, all store queue counters would be incremented for each store instruction. When the mapping of an instruction is determined, the corresponding counter is left unchanged but all other previously incremented counters in other clusters are decremented. Finally, when an instruction commits or is otherwise removed from the queue, the counter corresponding to its queue is decremented.

For each instruction that passes the dispatch stage, the optimistic technique increments only the counter corresponding to the predicted mapping. Once the mapping is known and the prediction turned out incorrect, the previously incremented counter is decremented and the counter corresponding to the correct mapping is incremented. When an instruction commits or is otherwise removed from the queue, its corresponding queue counter is decremented.

The hybrid scheme behaves like the optimistic scheme when the bank predictor is confident and like the conservative scheme when the bank predictor is not confident. The confidence predictor we use has the same structure and size as the meta predictor of the Tournament Predictor. It consists of a single table of 2048 saturated two–bit counters, which is indexed by the Program Counter of the instruction. Of the four possible values of each counter the two smaller counter values represent no confidence and the two greater values represent a confident result. For each misprediction the counter is decremented and for each correct prediction incremented. Since the confidence predictor is not used by other parts of the microarchitecture, it must be added especially for this scheme. This adds some overhead compared to the other two schemes that do not use the confidence of the predictor.

During a pipeline flush, care has to be taken to undo the changes on the counters of each flushed instruction correctly to guarantee correct behavior of the dispatch throttling mechanism. If flushes are implemented as we described in Section 4.1.12 it is sufficient to reset the counters to zero.

5.1.2 Dispatch Throttling: Evaluation

We compare the three FCD schemes with a baseline that does not contain any flow control scheme. First, we will examine the frequency of flush events for the four configurations. In our proposal, flush events are caused by four different types of events: memory dependency violations, memory queue overflows, memory queue deadlocks, and branch mispredictions.

Memory dependency violations cause flush events but are rare, first, because our configuration includes a store wait table as dependency predictor (see Section 4.2.3 of Chapter 4), and second, because we split store instructions so that store addresses are available early to avoid dependency violations (see Section 4.1.8 of Chapter 2).

Overflow events occur when the memory queue is full (see Section 4.1.5 of Chapter 2), but more instructions arrive at the memory unit. The most frequent events are deadlock events. Deadlock events occur when the memory queue is full and an instruction that is entering the memory unit is older than all instructions in the memory queue.

The frequencies of the flush events affect each other mutually. In addition, since we are interested in the overall effect of our proposals, the diagrams show the sum of all different types of flush events. We count flush events relative to 100 committed instructions to get numbers that are more tangible. This metric avoids distortions that might otherwise occur by

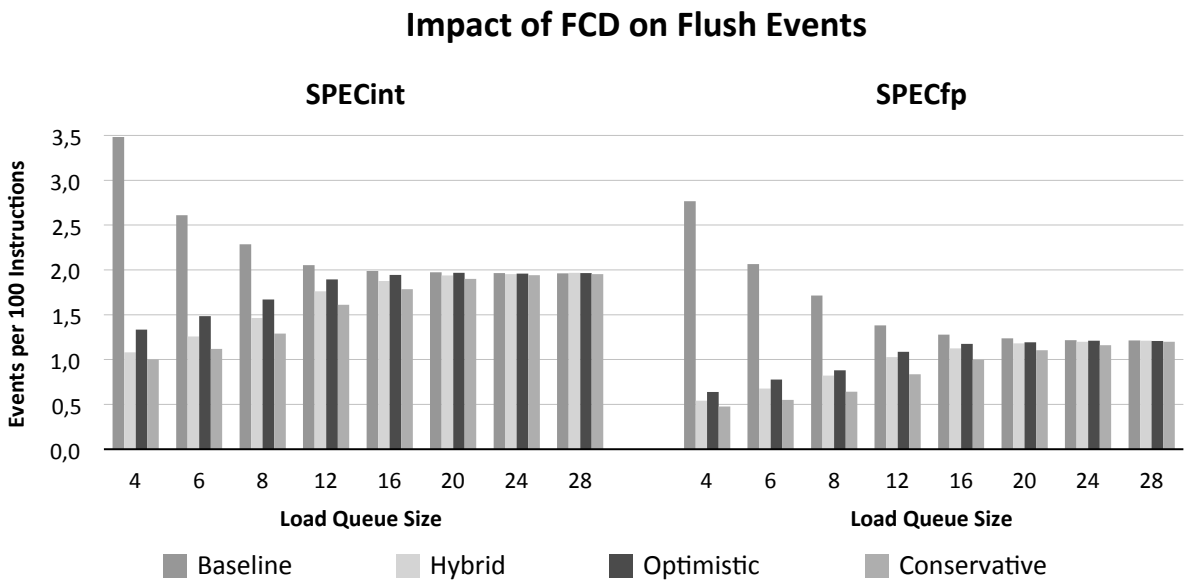

Figure 5.1: Impact of Dispatch Throttling Schemes on the Frequency of Deadlock Events. Events are measured in events per 100 committed instructions. The baseline configuration includes no throttling scheme.

considering dynamic instead of committed instructions. Because we are counting flush events, this is especially important. The size of the load queue has a major impact on the results and because the load queue size is an important cost factor, we include the load queue size as an explicit parameter in all of our graphs. We include smaller and larger than reasonable sizes to illustrate the impact of the load queue size. The figures show the number of entries per cluster, e.g. the number 16 denotes 16 entries in each of the four clusters or 64 entries in total.

Figure 5.1 shows the impact of the dispatch throttling schemes on the frequency of flush events. The integer benchmarks show a higher number of flush events than the floating-point benchmarks. The baseline behaves as expected. The number of flushes is reduced as the load queue is increased in size. This reduction is the result of less load queue overflow events.

However, the three dispatch throttling mechanisms show the opposite behavior. Starting from a lower level of flushes the number increases with load queue size. The dispatch throttling is especially severe for small load queues. Not only does it stop load instructions from dispatching, it also impedes the dispatching of all instructions that follow a stopped load instruction.

Adding throttling to the baseline turns it into a more conservative configuration that executes significantly less speculative instructions. As a result, all types of flush events are reduced. For the hybrid and optimistic control flow schemes, larger load queues lead to a more aggressive dispatching and consequently more flush events. The conservative scheme shows the same effect, but the increase in flushes is very small. Memory queue overflows and memory queue deadlocks are eliminated by the conservative control flow scheme, and since the dispatch rate is lower than for the other configurations, the number of the other flush

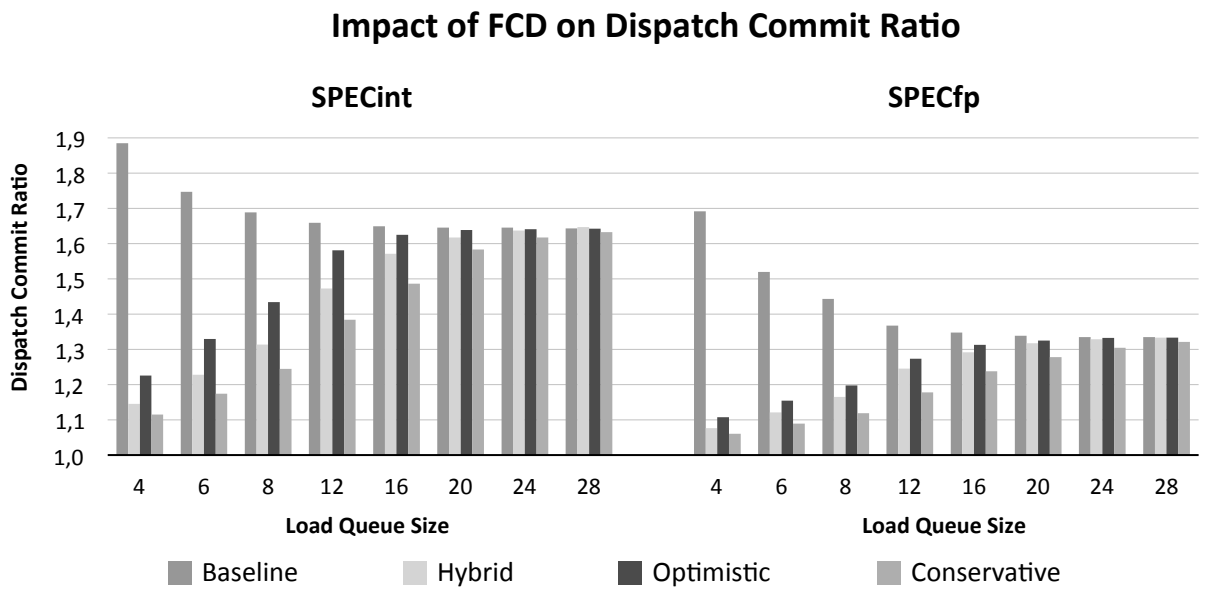

Figure 5.2: Impact of Dispatch Throttling Schemes on the Dispatch Commit Ratio. The Dispatch Commit Ratio is the number of dispatched instructions divided by the number of committed instructions. The baseline configuration includes no throttling scheme.

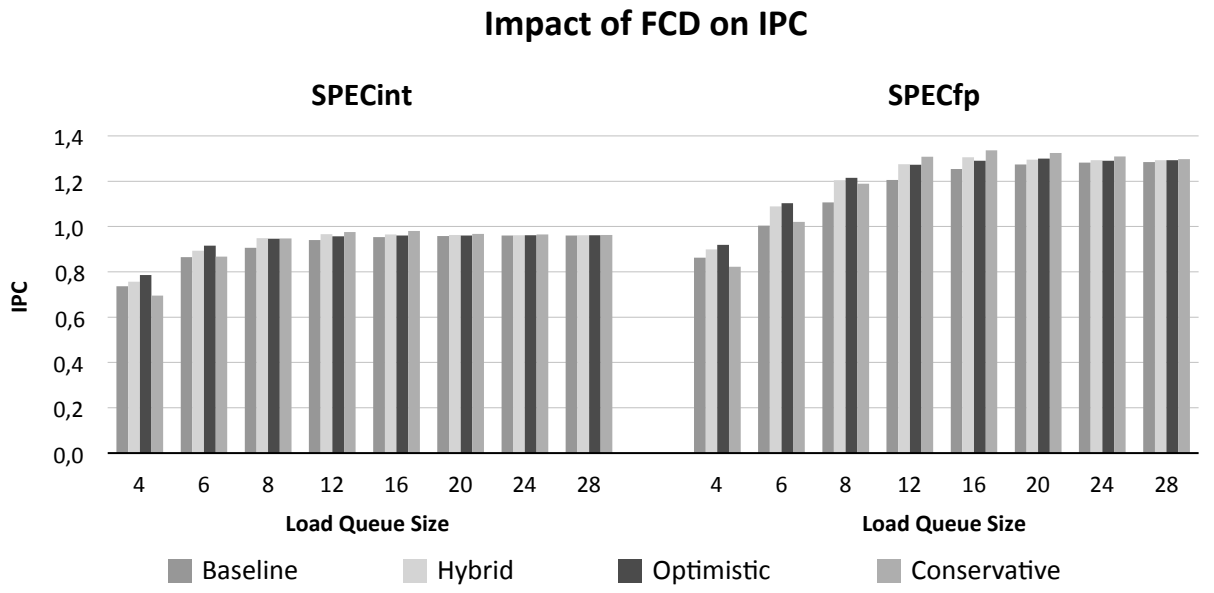

Figure 5.3: Impact of Dispatch Throttling Schemes on Performance. The baseline configuration includes no throttling scheme.

events is reduced too. (The other flush events are memory dependency violations and branch mispredictions.)

Next, we will look at the ratio between dispatched and committed instructions. This ratio is an indicator of the amount of speculative activity of the microarchitecture (and to some degree of the energy efficiency).

Figure 5.2 visually resembles very much the results from Figure 5.1. The most noticeable difference is the result of the conservative flow control scheme. While the number of pipeline flush events is hardly influenced by the load queue size, the dispatch commit ratio is changed significantly. This means that each pipeline flush evicts more instructions for the conservative flow control than for the other configurations. The frequent pipeline flushes of the other configurations likely shadow each other so that their overall negative effect is diminished.

Figure 5.3 illustrates the impact of the throttling schemes on performance. Neither scheme can convert the lower flush ratio into a significant performance advantage. The performance of configurations with very small load queues benefits from the control flow mechanisms. The performance gained by less frequent pipeline flushes is offset almost exactly by the performance lost due to the throttling of the dispatch stage.

An interesting anomaly in Figure 5.3 is the fact that the best result is not achieved by the most expensive configuration but by the conservative flow control with a load queue size of 16 entries. For small queues, the reduction in flushes helps performance but is constrained by excessive dispatch throttling. Larger queues don't suffer from excessive dispatch throttling but experience more flushes, which limit performance. In our experiments, a load queue size of 16 entries happens to offer the best tradeoff and performs better than the largest queue size (1.8% better for integer and 3.1% better for floating-point).

Even though the schemes presented in this section could not improve performance significantly (except for very small queues), they significantly reduce the speculative activity in the out-of-order core of the processor, which results in an improved energy efficiency.

5.2 Pre–Access Queues

We introduce pre–access queues in our memory unit to decouple the execution of memory instructions from the calculation of memory addresses (see Figure 5.4). This allows us to delay the execution of memory instructions until the required execution resources (namely a memory queue entry) are available. If the pre–access queues can hold enough instructions, we can avoid memory queue overflows and costly pipeline restarts.

The need for pre–access queues arises from overflows of the memory queues. Avoiding overflows is particularly important for small memory queues, which tend to overflow frequently. Growing the memory queue solves the problem but this remedy comes at a price. Memory queues contain expensive CAM structures and larger queues operate slower and consume more energy. The pre–access queue in comparison is a simple FIFO memory that is fast and consumes little energy.

We place the pre–access queue directly before the memory access stage in the memory pipeline. If an instruction cannot access memory (e.g. because the memory queue is full) instead of causing a pipeline flush the instruction is buffered in the pre–access queue.

Instructions that already reside in the pre–access queue have priority over instructions that arrive afterwards. Pre–Access queues operate at a different spot in the pipeline than the flow control techniques described earlier in this chapter. Both mechanisms can be used in combination to amplify their effect.

However, pre–access queues have some limits. Since they are FIFO memories, at any point in time there is only one instruction per queue that can issue to a memory pipeline. This lack of choice makes it difficult to block instructions because circular dependencies may be present, inhibit the queue from making further progress, and cause a deadlock. (We will look again at this problem when we introduce memory issue queues in Section 5.3) Therefore, we block the pre–access queue only on the condition that the memory queue is full. This allows us to check for only one deadlock condition that arises when the oldest instruction in flight is located in a blocked pre–access queue. Lastly, pre–access queues entries are no substitute for memory queue entries. A lack of memory queue entries will withhold instructions from accessing memory and therefore limit parallelism. In contrast, the primary function of the pre–access queue is to avoid costly pipeline flushes.

5.2.1 Pre–Access Queues: Implementation

Figure 5.4 shows how the different pre–access queues fit into the design. The load pre–access queue (LPQ) receives load instructions from the interconnection network (ICN), buffers the instruction if necessary, and finally issues the instruction for execution in the memory unit. The store pre–access queue (SPQ) works in a similar way for store address instructions (see Section 4.1.8 of Chapter 2). The store data pre–access queue (SDPQ) is not an independent

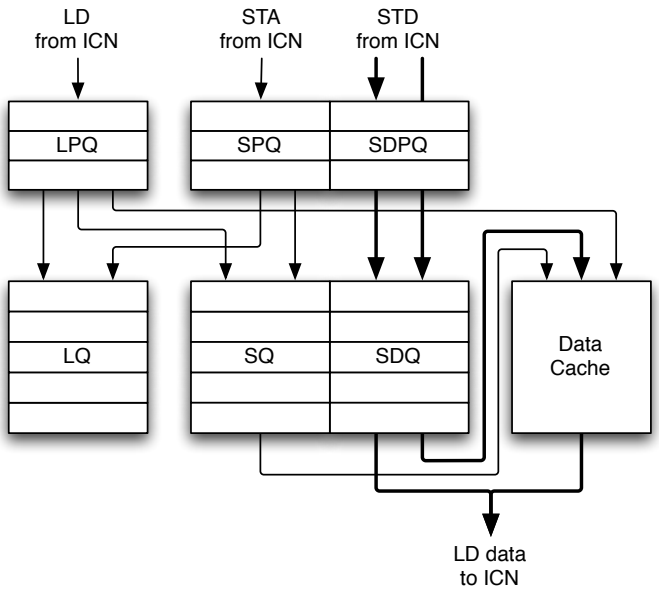

Figure 5.4: Distributed Memory Unit with Pre–Access Queues. These queues are the Load Pre–Access Queue (LPQ), the Store Pre–Access Queue (SPQ), and the Store Data Pre–Access Queue (SDPQ).

queue but essentially a buffer that holds store data instructions while their corresponding store address instruction is waiting in the store pre–access queue.

When there is no space left in the memory queue, instructions wait in the pre–access queue instead of causing an overflow. Overflows can still occur when the pre–access queue is full. However, since the pre–access queue contains no CAM structure it is far less problematic to increase its size. With a reasonable pre–access queue size, the number of overflows is significantly reduced. To decrease the number of overflows even further, the dispatch stage could stall when one of the pre–access queue fills up to a certain mark, e.g. half of its entries. However, we found that this optimization has no measurable effect on performance.

The pre–access queue also assumes the function of decoupling the inter cluster network (ICN) from the memory pipeline. To reduce the complexity of the memory units we limit them to execute no more than one load and one store instruction per cycle. This matches the bandwidth of the address generation units, which are limited in a similar way to calculate the addresses of only one load and one store instruction per cycle. However, when the distribution of instructions to memory pipelines is imbalanced, a bandwidth mismatch can occur at the local level. Each cycle up to four memory instructions may arrive at any given memory unit (one from each of the two links of the ICN—a bidirectional ring—and two from the address generation units of the local cluster) but no more than two instructions can execute every cycle. The pre–access queue substitutes the ICN network buffer to decouple the memory pipeline from incoming instructions. Therefore, they require a mechanism to decouple the network from the memory units.

Instructions that arrive from the interconnection network are always inserted into one pre–access queue, where they remain until they are successfully executed. After a successful execution, they are removed from the pre–access queue and inserted into the corresponding memory queue.

If the pre–access queue contains no instructions, any instruction that arrives from the interconnect network is executed immediately while it is inserted into the pre–access queue as well as the memory queue. This way any unnecessary delay in a potentially critical instruction is avoided.

Store data instructions that arrive at the memory unit are associated to their corresponding store address instructions, as described in Section 4.1.8 of the previous chapter.

5.2.2 Pre–Access Queues: Evaluation

We compare a configuration with pre–access queues, the baseline from Chapter 4, the conservative flow control scheme, and a combination of the conservative flow control scheme with pre–access queues. We will use the same methodology as in Section 5.1.2 before. The

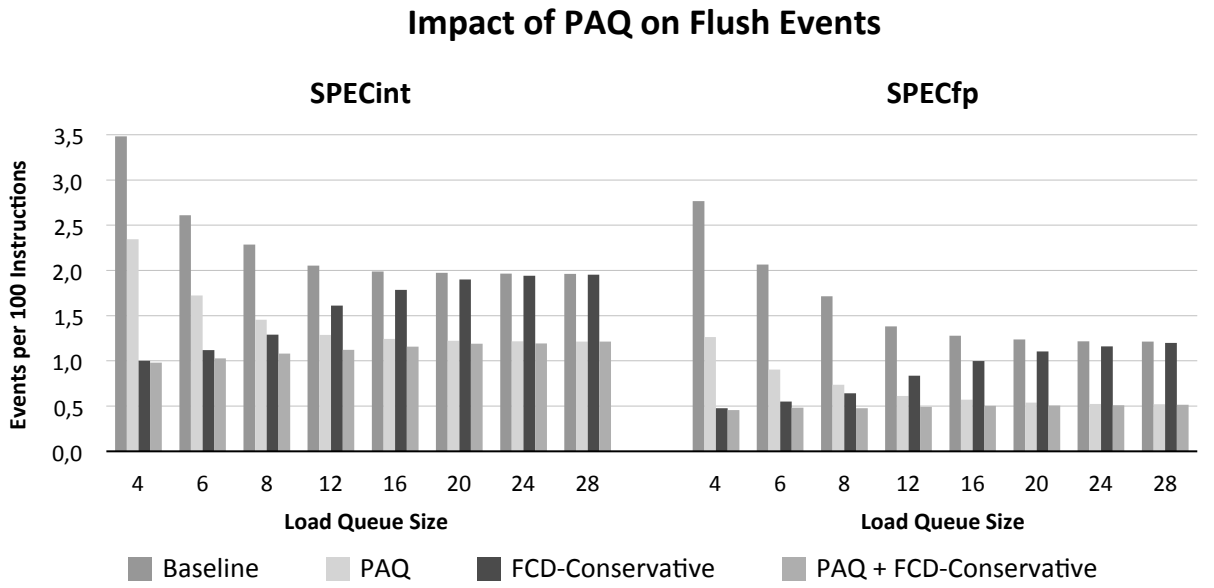

Figure 5.5: Impact of Pre–Access Queue on the Frequency of Deadlock Events. Events are measured in events per 100 committed instructions. The baseline configuration includes no pre–access queue or throttling scheme.

pre–access queues have a size of 16 entries. We found that larger pre–access queues to have no measurable impact on performance.

Figure 5.5 shows the frequency of flush events. The results for the baseline and conservative flow control scheme are the same as discussed previously in Section 5.1.2. The pre–access queue configuration shows consistently less flush events than the baseline. For load queues bigger than 8 entries it also shows less flush events than the conservative flow control scheme. The combination of pre–access queue and conservative flow control shows fewer flushes than any of the other configurations for all load queue sizes.

Comparing the four configurations at the extreme sizes, we observe that for small sizes the discerning parameter is the throttling mechanism, but for large sizes it is the presence of a pre–access queue.

Configurations with very small load queues suffer from many load queue deadlocks. The conservative control flow mechanism avoids all deadlocks in the load queue while a pre–access queue cannot inhibit this source of pipeline flushes. Consequently, the configurations with dispatch throttling show less overall flush events for small load queues.

Configurations with a rather aggressive dispatch policy (all configurations except conservative flow control with small load queues) are more likely to cause bandwidth imbalances that overflow the ICN network buffers (of two entries). The pre–access queue instead is larger than an ICN network buffer and does not overflow easily. Consequently the configurations with a rather aggressive dispatch policy depend primarily on the pre–access queue to reduce overall flush events.

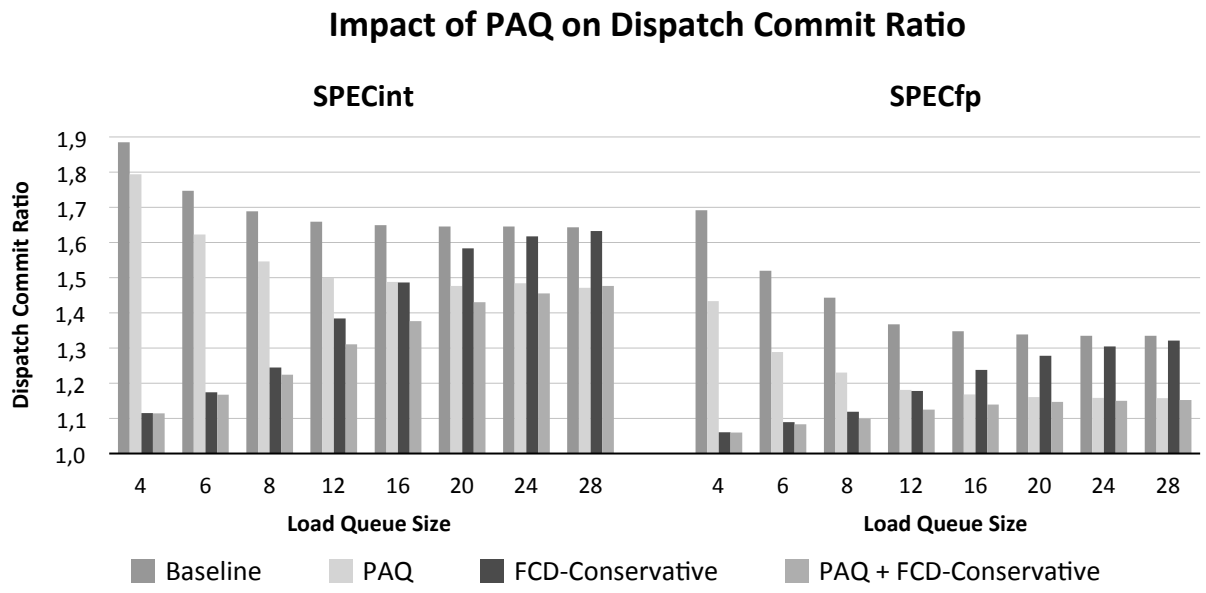

Figure 5.6: Impact of Pre–Access Queue on the Dispatch Commit Ratio. The Dispatch Commit Ratio is the number of dispatched instructions divided by the number of committed instructions. The baseline configuration includes no pre–access queue or throttling scheme.

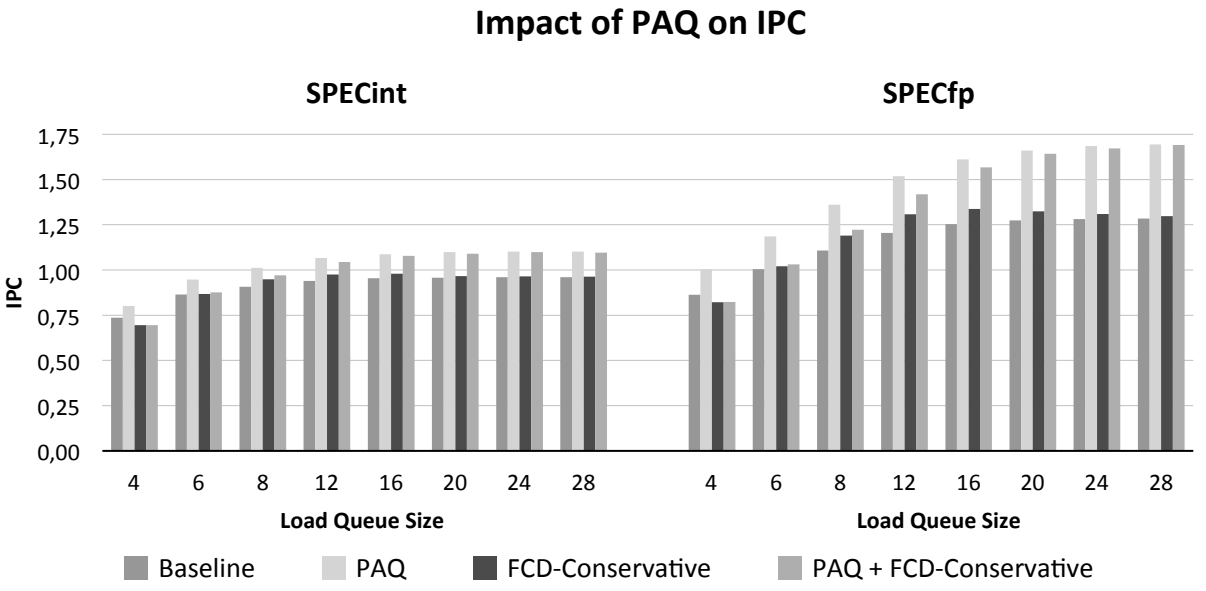

Figure 5.7: Impact of Pre–Access Queue on Performance. The baseline configuration includes no pre–access queue or throttling scheme.

Figure 5.6 shows the ratio of dispatched to committed instructions. The graphs resemble Figure 5.5 and the configuration that combines the pre–access queue with the control flow has the best dispatch commit ratio. The effectiveness of the control flow scheme diminishes with increasing load queue size.

Figure 5.7 shows the impact on performance. The configuration with pre–access queue but without dispatch throttling achieves the highest performance for all load queue sizes. The

performance improvement compared to configurations without pre–access queue is significant for all load queue sizes. The best configuration with a pre–access queue is 12,5% faster for integer and 26,7% faster for floating-point than the conservative dispatch throttling. The integer performance reaches a plateau at a load queue size of 20, floating-point performance at a size of 24 entries. This improvement in performance is the result of the reduction in flush events, which we observed earlier combined with a more aggressive dispatch policy compared to the flow control in dispatch.

The configuration with both, a pre–access queue and the dispatch throttling mechanism, eventually reaches the same performance plateau as the best configuration but its performance is inferior for most queue sizes. While this configuration is able to reduce flush events better than the other configurations, its less aggressive dispatch policy prevents it from exploiting as much parallelism as the best configuration.

In summary, the pre–access queue increases performance significantly for all benchmarks and queue sizes over the baseline and the flow control at dispatch. Floating-point performance is increased significantly, especially for large load queues.

In the next sections, we will propose alternative techniques that are more effective in avoiding deadlocks without sacrificing performance.

5.3 The Memory Issue Queue

We introduce memory issue queues in our memory unit as a refinement of the pre–access queues we presented in the last section. Like pre–access queues, memory issue queues decouple the execution of memory instructions from the calculation of memory addresses (see Figure 5.9). This allows us to delay the execution of memory instructions until the required execution resources (e.g. a memory queue entry) are available. If the memory issue queues can hold enough instructions, we can resolve resource conflicts without costly pipeline restarts. Furthermore, memory issue queues allow us to establish an issue policy that decides which instructions are chosen for issue. Issue policies not only can improve performance, they also provide a basis for a new set of techniques. These techniques selectively block instructions in the memory issue queue and thereby reduce the frequency of events that cause expensive pipeline flushes (e.g. deadlocks, memory dependency violations, etc.).

The introduction of the memory issue queues implies that memory instructions will now issue two times: first to calculate the address and then to access memory. The new queues allow memory instructions to reissue if an error occurs during their execution. Queue overflows are one example of such an execution error, but such errors also include for example partial memory dependencies (where several memory dependencies exist for a single instruction) and store–load dependencies, where the store data has not yet arrived and the load can therefore not yet execute successfully. If no mechanism for a reissue exists, a pipeline restart may be

the only alternative. Providing a reissue scheme allows the architecture to handle these cases more efficiently.

Some commercial microarchitectures allow instructions to reissue directly from the memory queue. Examples are the microarchitectures of the P6 [Abr97] (see Section 2.2.3 of Chapter 2) and the Athlon [AMD02] (see Section 2.2.4 of Chapter 2). In these cases, the memory issue queue and the memory queue are the same structure, sharing the functionality of both queues. However, the expensive functionality of a CAM is only required for entries that hold instructions that did already execute successfully, since only those entries need to be searched for potential memory dependency violations. On the other hand, once successfully executed, these instructions do not need to be reissued again; hence they do not require the functionality of an issue queue. By providing two separate, specialized structures for searching instructions and for reissuing them, we aim to reduce costs.

The memory issue queue is also the optimal point in the pipeline to enforce issue policies for load and store instructions, e.g. giving precedence to older instructions to improve performance. Because the memory issue queue can issue any instructions, it can also selectively block instructions without blocking the whole queue (and accidentally create a deadlock in the memory issue queue). This can e.g. be used to avoid deadlocks in the memory queue (see Section 5.4). While these policies can be implemented in earlier stages of the pipeline, doing so would unnecessarily create a delay between the decision to issue and the actual issue, and thereby increase the latency of the delayed instructions. This delay is minimized by enforcing the issue policy as late as possible–in the memory issue queue.

5.3.1 Memory Issue Queue: Implementation

The memory issue queue is a refinement of the pre–access queue. It serves the same purpose but adds functionality. The main difference is that the memory issue queue is not a FIFO and therefore the management of queue entries is slightly more complex. Instructions enter the memory issue queues as soon as their addresses are calculated by the out-of-order core and leave the queue once they successfully issue. Therefore, instructions are inserted into and removed from the queue out of program order. This suggests a similar spatially unordered implementation as memory queues, i.e. allocation and release of queue entries could be solved in a similar manner. (See Sections 4.1.4 and 4.1.6 of Chapter 4. Spatially unordered issue queues have also been proposed by Buyuktosunoglu et al. [Buy02].)[18]

Instructions that arrive from the interconnection network are always inserted into one memory issue queue where they remain, until they are successfully executed. The memory issue queue selects a ready instruction in the queue to be issued in the next clock cycle. After

[18] Other organizations are possible. E.g. instructions could be kept in the order by which they were inserted and the queue could be compacted when an instruction is released.

a successful execution, they are removed from the memory issue queue and inserted into the corresponding memory queue.

If the memory issue queue contains no instructions that are ready to issue, any instruction that arrives from the interconnect network is executed immediately while it is inserted into the memory issue queue as well as the memory queue. This way any unnecessary delay in a potentially time critical instruction is avoided. This situation might occur when the issue queue is empty or all instructions in the issue queue are blocked (e.g. to avoid a deadlock, see Section 5.4).

Wakeup instructions for reissue

Instructions, which did not execute with success, remain in the memory issue queue, but are marked as not ready. The queue includes a wakeup mechanism to promote these instructions again to the ready state. A simple option is to use a small timer per instruction and promote instructions after a fixed number of clock cycles regardless of the event that prevented them from executing correctly. Another option is to include a wakeup scheme where each instruction monitors an event bus to detect events that indicate the absence of the condition that prevented its successful execution. This second option is similar to the wakeup phase in a conventional issue queue where instructions monitor an event bus (instead of a result bus) for event tags (instead of register tags). [Abr97]

Select instruction for issue

While it is not necessary for the correct function of the memory unit to select always the oldest ready instruction for issue, this policy improves performance because older instructions are more likely to be in the critical program path. Figure 5.8 shows how a mechanism to identify the oldest ready instruction in the queue might work.[19] While the memory queue itself is unordered the illustrated mechanism uses a bit vector indexed by the global load or store sequence number to represent the ready state of each instruction in the queue. This spatial ordering allows a straightforward identification of the oldest ready instruction. Whenever the ready state of an entry in the unordered memory queue changes the ready bit vector is updated accordingly. Initially all instructions are ready and only if issue is aborted due to a conflict they pass into the not ready state.

The transition from the not ready state to the ready state is triggered by the wakeup logic and is more complex. More than one instruction may pass from the not ready to the ready state in a single cycle (in an extreme case all instructions in the queue might pass from not

---

[19] Other issue policies than *oldest first* are possible. E.g. instructions could be issued in the order by which they were inserted into the queue. Such a policy would be cheaper to implement, but performance would be slightly lower. Whatever the issue policy, the important difference is that unlike a FIFO queue an issue queue allows ready instructions to issue even if they are not at the tail of the queue (i.e. even if they are not the instruction, which spent most time in the queue).

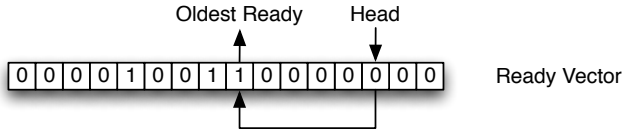

Figure 5.8: Selecting the Oldest Ready Instruction. A ready vector dedicates one bit to each instruction in flight and is indexed by the global load or store sequence number. The head indicates the oldest memory instruction in flight. The oldest ready instruction is found by scanning from the position of the head and skipping all adjacent 0 bits until the first 1 bit is found. This bit indicates the oldest ready instruction in the queue. The process of scanning is similar to a carry propagating during addition.

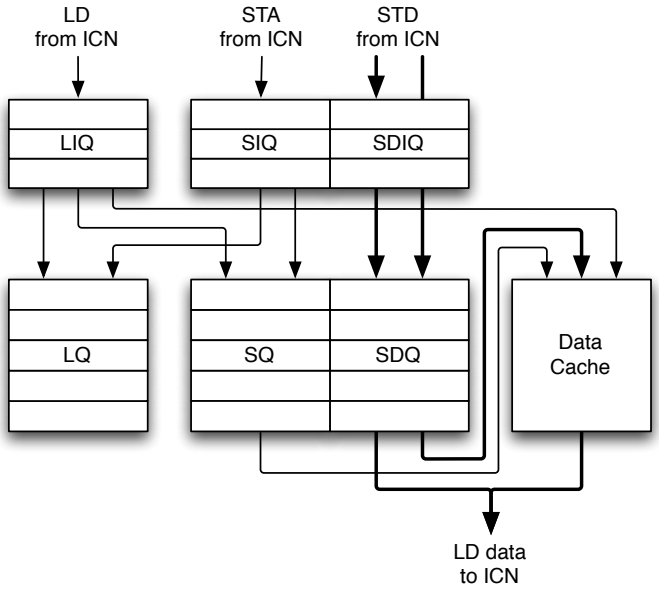

Figure 5.9: Distributed Memory Unit with Memory Issue Queues. These queues are the Load Issue Queue (LIQ), the Store Issue Queue (SIQ), and the Store Data Issue Queue (SDIQ).

ready to ready in a single cycle). Instantly updating the ready vector from the unordered memory queue could be challenging because the bit position in the ready vector corresponding to each woken instruction must be identified. An update could be accelerated by caching the decoded bit–vector of the global sequence number for each memory queue entry.

However, the update does not have to be instant to guarantee correct operation of the queue and the extreme case of waking up the entire queue is very infrequent. A simpler solution to update the ready vector gradually might be preferable; e.g. the global sequence number of the two instructions can be decoded with two simple decoders, which could iteratively update the entire queue.

Similar to pre–access queues store data instructions that arrive at the memory unit are associated to their corresponding store address instructions, as described in Section 4.1.8 of Chapter 4.

5.3.2 Memory Issue Queue: Evaluation

To evaluate the effectiveness of the memory issue queue we will first look at the frequency of pipeline flush events. Pipeline flushes are an important measure because they do not only affect performance negatively, but also waste energy executing instructions multiple times.

In the context of the load queue, we distinguish three types of flush events: memory dependency violations, load queue overflows and load queue deadlocks. Memory dependency violations cause flush events but are rare, first because our configuration includes a store wait table as dependency predictor (see Section 4.2.3 of Chapter 4), and second because we split store instructions so that store addresses are available early to avoid dependency violations (see Section 4.1.8 of Chapter 2). Overflow events occur when the memory queue as well the memory issue queue (or the pre–access queue, see Section 4.1.5 of Chapter 2) are full, but more instructions arrive at the memory unit. Deadlock events occur when the memory queue is full and the oldest instruction in the memory unit is blocked in the memory issue queue (or the pre–access queue).

Figure 5.10 shows the frequency of flush events for configurations with a load issue queue of 16 entries each. For comparison, we also show a pre–access queue of the same size and combinations with the conservative flow control dispatch mechanism.

Comparing the two configurations without flow control at dispatch, the configuration with the load issue queue suffers less flush events than the configuration with the pre–access queue for all simulated load queue sizes. A full load queue is a precondition for a deadlock flush, and the flush frequency does indeed decline with increasing queue size for both configurations. Another precondition for a deadlock are younger instructions that occupy all

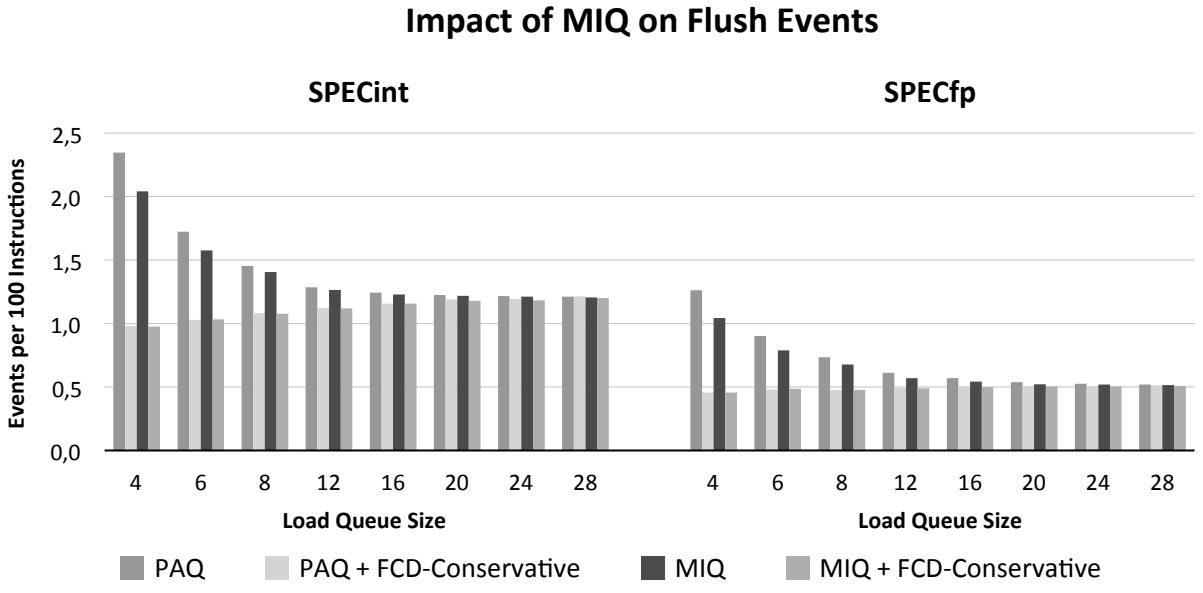

Figure 5.10: Impact of Load Issue Queue on Flush Events. Events are measured in events per 100 committed instructions. Only the arithmetic mean is shown for each queue size.

load queue entries while an older instruction is still waiting to issue. Because the load issue queue gives older instructions a higher issue priority than younger instructions, it reduces the probability of a deadlock. The pre–access queue issues instructions strictly in incoming order with no respect to age and is therefore more likely to create a deadlock.

The combinations with the conservative flow control dispatch mechanism behave virtually identically. The number of flush events increases slightly for larger queues. For large queue sizes, the number of flushes is similar for all four configurations, for all other queue sizes the flow control at dispatch produces significantly less flush events.

Figure 5.11 shows the dispatch commit ratio as a function of queue size. The results resemble very much Figure 5.10. The memory issue queue improves the dispatch commit ratio especially for small load queues.

Figure 5.12 shows the impact of the load issue queue on performance as a function of queue size. Comparing to the earlier Figure 5.10 we can observe that for the memory issue queue the reduction of the flush frequency translates directly into an improved performance for all queue sizes and for integer as well as for floating-point benchmarks. The configuration with a load issue queue but without dispatch throttling reaches a 1.6% higher plateau at a load queue size of 20 entries for integer and a 3.1% higher plateau at a size of 24 entries for floating-point benchmarks.

As already observed in Section 5.1.2 for the flow control at dispatch the reduction in flush events does not directly translate into higher performance. The reduction of flushes comes at the price of executing significantly less instructions speculatively. This is especially evident for small queues.

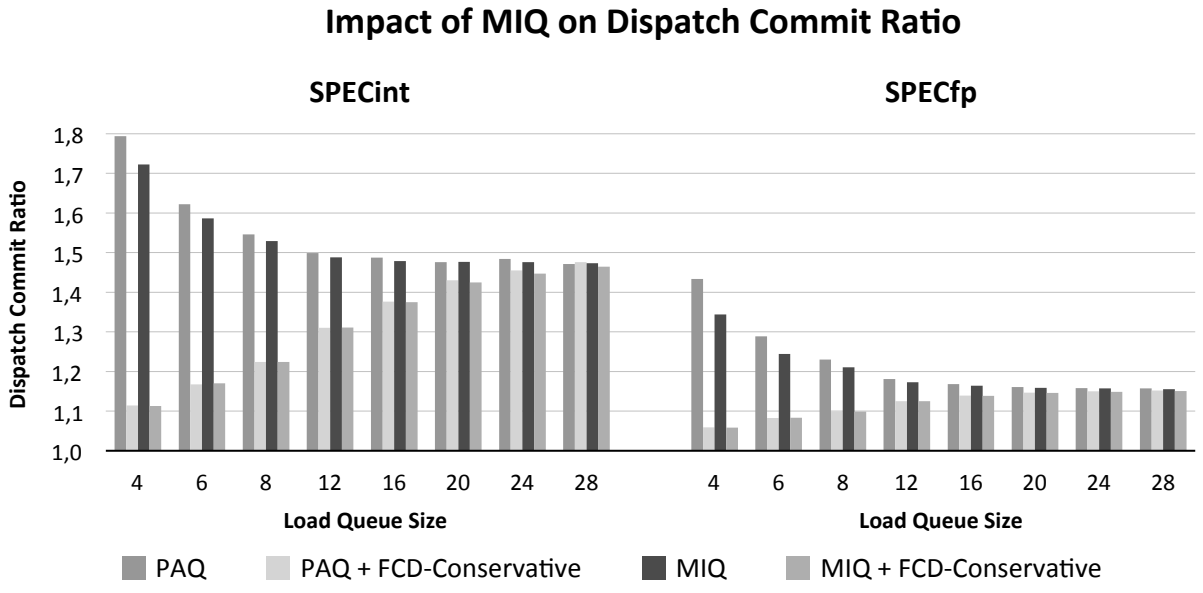

Figure 5.11: Impact of Load Issue Queue of 16 entries on the Dispatch Commit Ratio. Only the arithmetic mean is shown for each queue size.

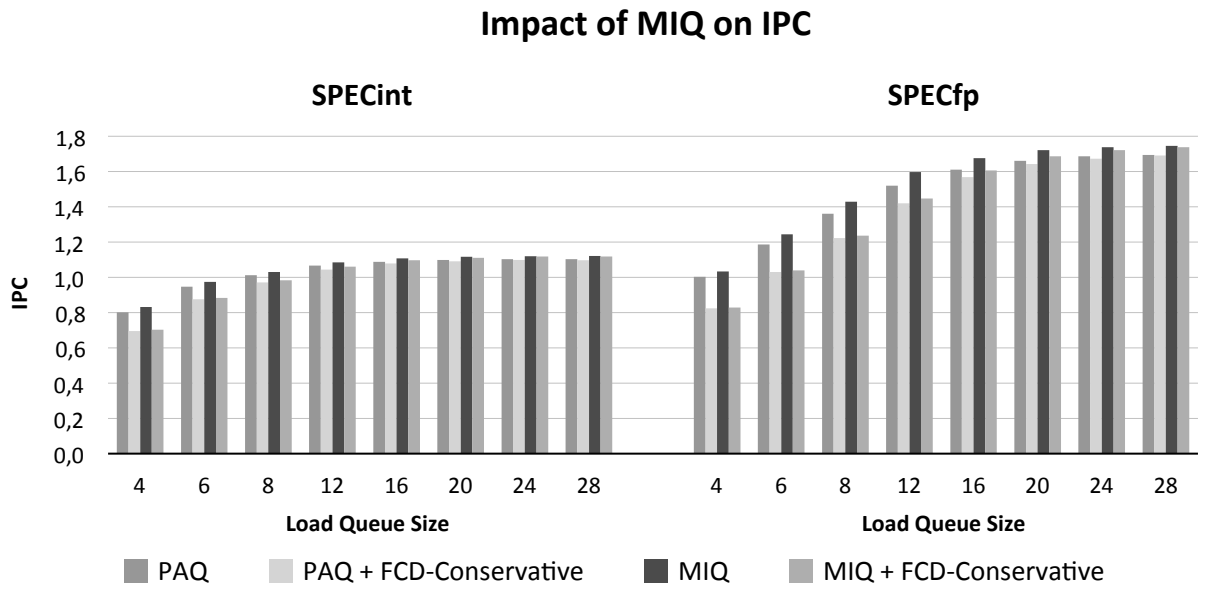

Figure 5.12: Impact of Load Issue Queue of 16 entries on performance. Only the harmonic mean is shown for each queue size.

The memory issue queue can reduce the number of deadlock events and thereby improve performance. However, especially for smaller load queue sizes deadlocks still occur frequently and reduce performance. This is the motivation for another mechanism proposed in the following section. This mechanism will eliminate deadlocks without compromising performance, as conservative flow control at dispatch does.

5.4 Conservative Deadlock Aware Entry Allocation (CDA)

The memory issue queue presented in the last section enables us to choose an issue policy. In this section we describe an issue policy that allows us to avoid deadlocks altogether. This mechanism relies on information that is exchanged between the clusters over the interconnection network.

To avoid deadlocks completely an instruction may issue only, if all older instructions are guaranteed to be able to issue too. This includes instructions whose addresses are not yet calculated. Once instructions calculate their address and hence their mapping to clusters becomes known, they have to inform all clusters of the mapping. So to decide whether or not to issue an instruction, all instructions with unknown mappings and all instructions that map to the same cluster are taken into account. If the number of free entries exceeds the number of older instructions for the same cluster or with undetermined mapping, then it is completely safe to issue an instruction.

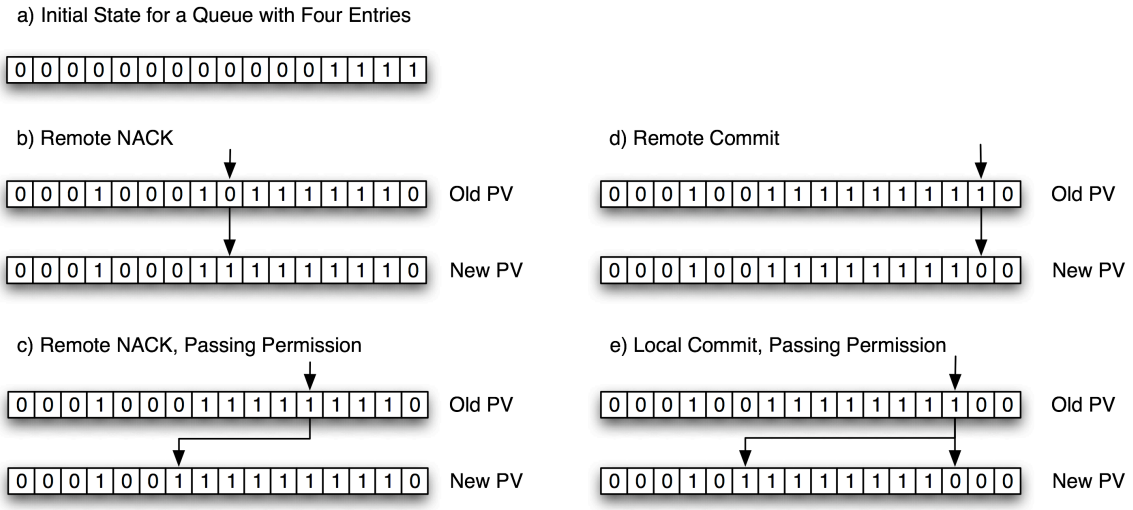

Figure 5.13: Updates of the Permission Vector (PV).

5.4.1 Conservative Deadlock Aware Entry Allocation: Implementation

The deadlock–aware entry allocation mechanism is built around a permission bit–vector (PV). This vector is indexed by the load sequence number or store sequence number and contains one bit for each dispatched, in-flight load instruction. (In the following discussion we will refer to the load queue only, however the same arguments apply to the implementation of the mechanism for the store queue.) Each memory issue queue has its own permission vector and each load instruction in-flight in the out-of-order core owns a bit in the permission vector. An instruction is allowed to issue, if its corresponding permission bit is set. The initial state after reset is a permission vector that contains all zeros except for the first N_{queue} positions, where N_{queue} refers to the size of the load queue. Figure 5.13 a) shows an example for a queue size of four. This configuration allows the four oldest instructions in-flight to issue but denies issue to all other instructions.

As more instructions are executed, the more information is known about the mapping of load instructions to clusters. Whenever a load instruction calculates its address, the instruction is sent to the corresponding cluster. Additionally, a NACK message containing the load sequence number is sent to all other clusters to inform them about the new mapping. Upon receiving the messages, the other clusters update their permission vectors in the following way: Figure 5.13 b) shows the example of a NACK message that is received for an instruction that had no prior permission to issue. In this case, its corresponding permission bit is set in the vector. Figure 5.13 c) shows the case of a NACK for a load that did have prior issue permission. In this case, the permission is passed along to the oldest instruction without issue permission. This is accomplished by passing the permission bit to the left until the first zero-bit is encountered, somewhat similar to carry propagation.

The number of one–bits in the permission vector at any time is equal to the number of received NACK messages plus N_{queue}. The mechanism gives issue permission to the first N_{queue} instructions out of all instructions that did not send a NACK message. Notice that the instructions that did not send a NACK message are exactly those, which are known to map to the local cluster or whose mapping is not yet known. By choosing the first N_{queue} instructions out of these instructions, deadlocks are completely avoided as we explained earlier.

To allow continuous operation the permission vector is organized as a circular buffer and the permission bits may pass from the leftmost to the rightmost bit. Upon commit of an instruction the corresponding permission bit is cleared in the remote clusters as shown in Figure 5.13 d). If the commit liberates an instruction in the local load queue, the next youngest load is given permission to issue, see Figure 5.13 e). Again, this is implemented by a carry–alike mechanism that traverses the permission bits to the left until a zero bit is encountered.

To guarantee the correct behavior of this mechanism, the size of the permission vector is defined to be equal to the maximum number of load instructions in-flight in the out-of-order core *plus* N_{queue} bits. These N_{queue} additional bits are necessary to handle extreme cases, for example, when a single cluster receives NACKs for all in-flight loads.

Since all clusters can calculate load addresses in parallel, there can be many NACK messages generated in a single cycle. To limit the complexity of the deadlock–aware entry allocation we limit the generation of load addresses in our experiments to one address per cluster per cycle. The generation of store addresses is also limited in the same way. The bandwidth of the address generation is matched by the cache bandwidth of one load and one store per cluster.

5.4.2 Conservative Deadlock Aware Entry Allocation: Evaluation

Figure 5.14 shows the impact of the technique on the frequency of flush events. For integer as well as for floating-point, the number of flush events is reduced significantly. Unlike the other configurations, the number of flushes for the conservative deadlock aware entry allocation does hardly depend on the size of the load queue. Since queue overflow events are handled by the memory issue queue and deadlocks by the allocation mechanism, these two sources of flush events are practically eliminated. The remaining sources—memory dependency violations and branch mispredictions—are mainly independent of the load queue size.

Figure 5.15 shows the dispatch commit ratio for the conservative deadlock aware entry allocation. Like the number of flush events, it is hardly affected by the size of the load queue. The conservative deadlock aware entry allocation shows a better ratio than all other configurations, the advantage is especially significant for small queue sizes.

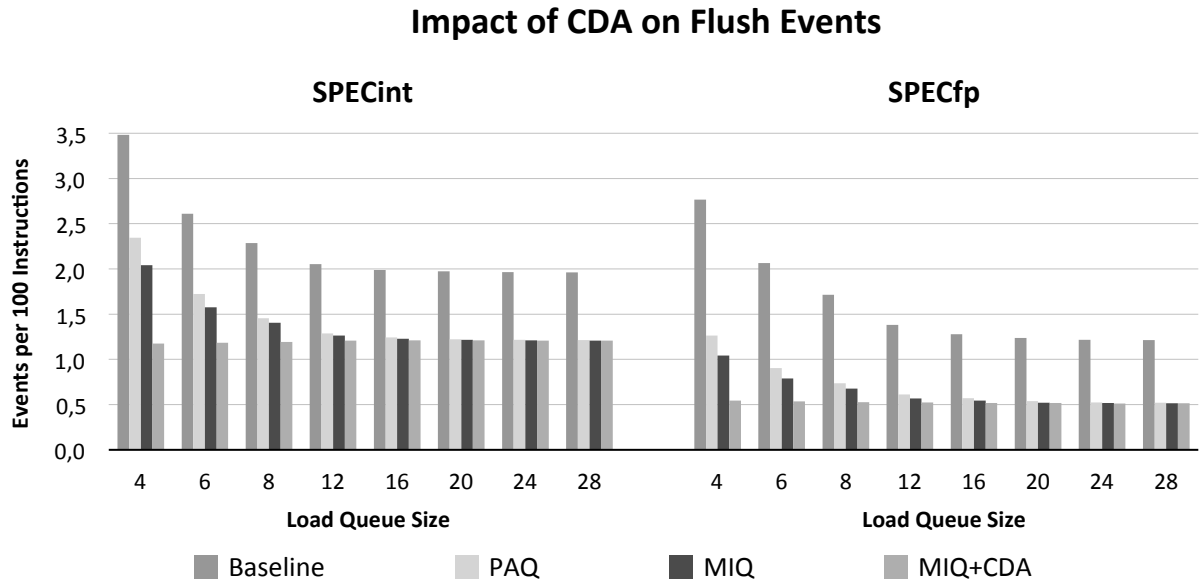

Figure 5.14: Impact of Conservative Deadlock Aware Entry Allocation on Flush Events. Only the arithmetic mean is shown for each queue size.

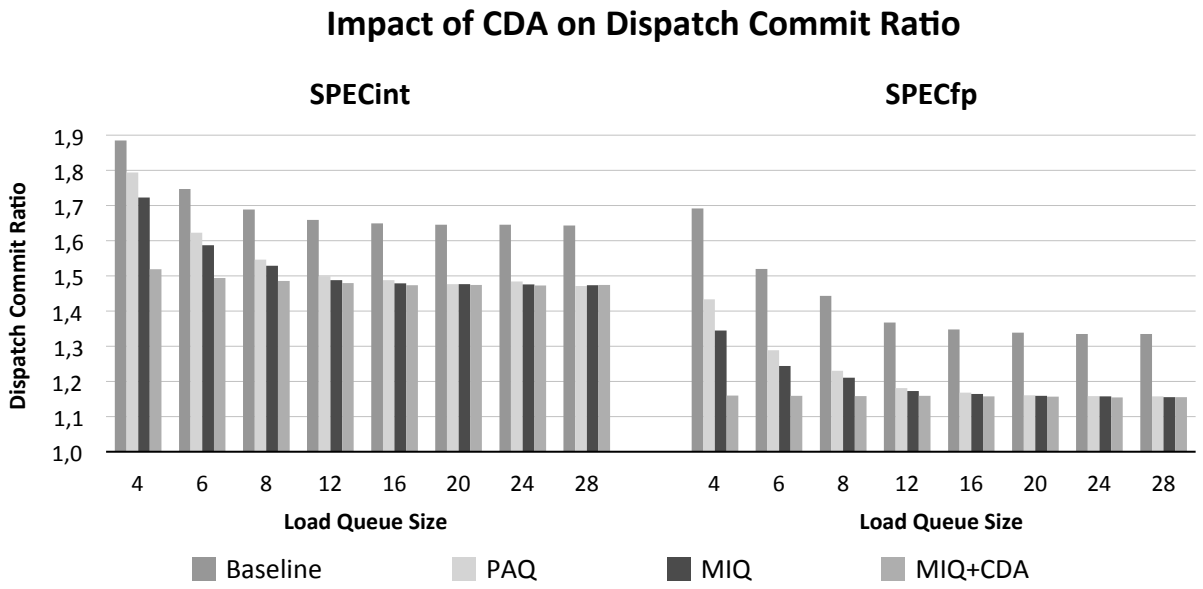

Figure 5.15: Impact of Conservative Deadlock Aware Entry Allocation on Dispatch Commit Ratio. Only the arithmetic mean is shown for each queue size.

Figure 5.16 shows the IPC as a function of load queue size. The technique has a significant effect on the performance of the configurations with small queues. The performance of configurations with large queues reaches a plateau and improvements are smaller. This is the expected result, because large queues incur very little deadlock events, so there is little potential for this technique. Little queues on the other hand suffer from deadlocks and benefit from conservative deadlock aware entry allocation. Nonetheless, the performance plateau is reached with smaller load queues (16 entries for integer and 24 entries for floating-point). A

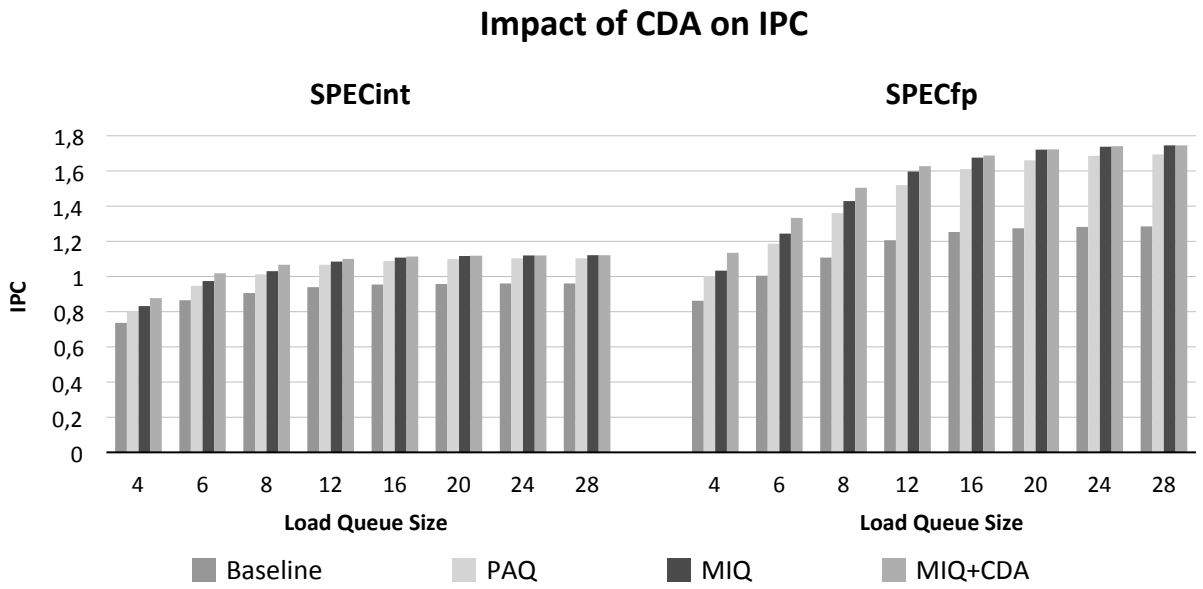

Figure 5.16: Impact of Conservative Deadlock-Aware Entry Allocation on Performance. Only the harmonic mean is shown for each queue size.

load queue size of 12 entries can achieve integer performance within 1.9% of the largest load queues and with 20 entries within 1.3% of the best floating-point performance.

5.5 Early Release of Load Queue Entries

To maximize the effective load queue size, instructions should occupy load queue entries no longer than required. Traditionally, instructions liberate their load queue entries when they commit. By liberating entries earlier, we can make better use of the load queue. Depending on the design in question the load queue can serve different purposes. The two most common uses are the detection of data dependency misspeculations and the enforcement of memory consistency [Gha95] in the context of multiprocessors.

Data Dependency Speculation

Memory units that use data dependency speculation usually employ the load queue to detect misspeculations. These misspeculations are store–load ordering violations, which occur when a load was reordered with respect to a store and both instructions access a common memory location. This verification is required to guarantee correct memory semantics.

Example 5.1 illustrates the problem. Because the load instruction follows the store instruction in the original program order and both instructions access the same memory location, the value in register r0 must be the same as the value of register r1 at the end of the instruction sequence. However, if the out-of-order core decides to execute the load instruction before the store instruction, a stale value will end up in register r0. Later, when the store instruction is executed by the out-of-order core (more precisely: when it is inserted into the

Before: register r1 contains value one,
memory location (r2) contains value zero.

```
stl r1,(r2)   ; executes later
ldl r0,(r2)   ; executes first
```

After: register r0 contains value zero. (incorrect)

Example 5.1: Read-after-Write Dependency. The reversed order of execution and the aliasing of the two instructions lead to an incorrect result.

store queue), it will search the load queue and encounter the load instruction that is both younger and accesses the same memory location.

In our architecture, a pipeline flush will force the load and all instructions that depend on it to re-execute with the correct register values. A more sophisticated (and more complex) architecture might choose to selectively re-issue only those instructions that depend on the erroneous result of the load instruction. [Rot97]

Multiprocessor Memory Consistency

Shared memory multiprocessor systems may require further verifications of the memory ordering. Exactly which verifications are required, depends on the memory consistency model used for the multiprocessor system in question. The following examples illustrate the involvement of the load queue.

Before: register values identical on both processors,
register r1 contains value one,
memory location (r0) contains value zero.

Processor 1	Processor 2
`stl r1, (r0)`	`A: ldl r2, (r0)`
	`B: ldl r3, (r0)`

After: registers r2 and r3 of processor 2 contain values one and zero respectively. (incorrect)

Example 5.2: Load Ordering. The reversed order of execution of the load instructions, the execution of the store instruction inbetween and the aliasing of the three instructions lead to an incorrect result.

Example 5.2 shows an instruction sequence that violates the memory ordering model of all architectures we considered, Alpha [Alpha], x86 [IA32], Itanium [IA64M], SPARC TSO [Wea94], PowerPC [Fre05], and IBM z/Architecture [ESA390].

The two load instructions A and B access the same memory location. In this example, the two load instructions were executed out of program order. Instruction A observes the memory value written by processor 1 while instruction B observes the original memory value. This combination of results is not allowed by memory ordering models.

The situation described above in Example 5.2 occurs only if two load instructions access the same memory location and execute out-of-order. Therefore, some microprocessors like the Alpha EV6 [Kes98] and the Power4 [Ten01] search the load queue to detect this condition.

Before: register values identical on both processors,
register r0 contains the value one,
memory locations (r1) and (r2) contain value zero.

Processor 1	Processor 2
`stl r0, (r1)`	`A: ldl r3, (r2)`
`stl r0, (r2)`	`B: ldl r4, (r1)`

After: registers r3 and r4 of processor 2 contain values one and zero respectively. (incorrect for some architectures)

Example 5.3: Load/Store Ordering. The reversed order of execution of the load instructions, the execution of the store instructions inbetween and the aliasing of the four instructions lead to an incorrect result.

Example 5.3 describes another case that violates the memory ordering scheme of some architectures (x86, IBM z/Architecture, Itanium, SPARC TSO) but not the model of others (Alpha, PowerPC).

Here instructions access two different memory locations. Load instruction A observes the value written by processor 1 while load instruction B observes the original memory value. Again, the load instructions A and B were executed out of program order.

This problem can be circumvented by intercepting snoop requests that conflict with instructions, which were executed speculatively out of program order. These instructions are typically held in the load queue. In the example above, processor 1 will send snoop requests for both stores to the system memory bus. Processor 2 will receive these snoop requests and search its load queue for any speculatively executed load instructions that access the same memory location. If any matching loads are found these loads and any dependent instructions are re-executed and the incorrect result illustrated in Example 5.3 is thereby avoided. It should be noted that this technique also solves the problems described in Example 5.2.

Which mechanism is preferable depends on the memory ordering model in question. E.g. architectures like x86, IBM z/Architecture, Itanium, and SPARC TSO would implement the solution of Example 5.3. For other architectures like Alpha and PowerPC, the solution of Example 5.2 is sufficient. The solution of Example 5.3 will also give correct results for these two architectures, but will hurt performance because more instructions will be re-executed than is actually necessary.

Notice that the mechanism from Example 5.2 can be refined by combining it with the mechanism from Example 5.3. Considering Example 5.2, a pipeline flush is only required if a load instruction is both hit by an older load instruction and an external store instruction. This refinement increases the implementation complexity slightly, but avoids the performance loss of unnecessarily re-executed instructions. [Ten01]

Until now, we assumed that load queue entries are allocated when the load instruction is speculatively executed and released when the load instruction commits (or a pipeline flush occurs). However, the time frame during which a load instruction must be present in the load queue is often shorter.

Consider Example 5.1 of the last section: Let's assume a load instruction, which is present in the load queue. Suppose that all store instructions, which are older than this load instruction, have already searched the load queue for conflicts. Then this load instruction can no longer cause store–load conflicts and does no longer need to participate in these searches.

A similar argument holds for Example 5.2. Let's assume a load instruction, which is present in the load queue. If all load instructions, which are older than this load instruction, did already search the load queue, then this load instruction can no longer cause a conflict and does no longer need to participate in these searches.

These two observations enable us to propose a scheme to remove load instructions from the load queue without impacting the functionality of the queue. We call this scheme early release of load queue entries or ERLQ for short. The scheme can be applied to conventionally centralized as well as to distributed memory pipelines.

In case of a centralized memory pipeline a load instruction must comply to the following condition: All load and store instructions, which are older than the load in question, did already search the load queue. In a typical microarchitecture, load and store instructions search the load queue as they are inserted into the load and store queues. In this case, the condition is equivalent to the condition that all older load and store instructions are either present in the load or store queue or were already released or committed.

In case of a distributed memory pipeline, we must also consider instructions that are mapped to other pipelines. The condition to release a load queue entry is modified as follows:

All load and store instructions which are older than the load in question did already search the load queue or are known to execute in other pipelines.

The technique used to resolve Example 5.3 (intercepting snoop requests) requires the presence of all load instructions in the load queue. However, it seems reasonable to assume that snoop events compared to local stores are rather infrequent. (The appropriateness of this assumption depends on the nature of the software used and the scale of the multiprocessor.) If one structure is used to disambiguate local load instructions and another structure for external snoop requests the technique outlined above could still be applied to the first structure. Only the less frequently accessed structure, which handles external snoops, would have to accommodate all loads up to the commit stage.

5.5.1 Early Release of Load Queue Entries: Implementation

The early release mechanism checks periodically if the oldest instruction in the load queue meets the necessary conditions to be released early from the queue. There are two conditions to be met. First, all in–flight load instructions that are older than the candidate must either map to another cluster or have been already released from the load queue. This guarantees that there are no outstanding address comparisons for the candidate in question. Second, all store instructions older than the candidate must either map to another cluster or be present in the store queue (i.e. the addresses of the stores are known and all possible dependencies with the candidate load have already been resolved). If these conditions are fulfilled the candidate load is removed from the load queue.

The mechanism for the early release of load queue entries is built around two bit–vectors called the *allocation vector* and the *unknown vector*. There are two sets of these vectors in each cluster, one for the store queue and one for the load queue. Each load and store instruction in the out-of-order core owns one bit in each vector. This bit is accessed by indexing the vector with the load (or respectively store) sequence number.

To find the oldest load in the local load queue we use an allocation vector, which is indexed by the load sequence number just as the vectors in the previous section. The load queue allocation vector contains a one–bit if the corresponding instruction is present in the local load queue or a zero–bit otherwise. Starting from the head of the allocation vector (the head corresponds to the oldest in-flight load in any cluster) the vector is searched to the left until a one–bit is found. This bit marks the oldest instruction already present in the load queue. This procedure is illustrated in Figure 5.17 a).

Once the oldest load instruction in the queue is identified, the mechanism has to check for instructions that are older than the load in question but are not present in the queue. Figure 5.17 b) illustrates the process of identifying these unknown predecessors. In a step similar to the one used to identify the oldest load, a predecessor mask is generated. This mask marks all predecessors with a one–bit in the corresponding position. This step may be performed in

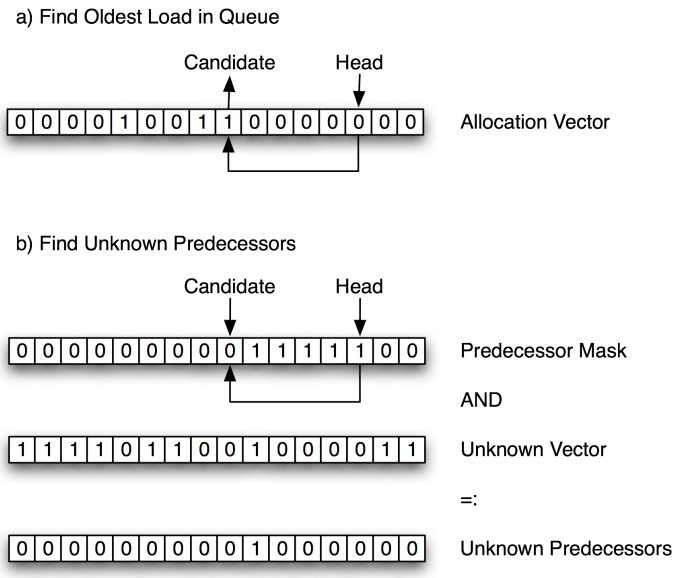

Figure 5.17: How to Find a) the Oldest Load and b) Unknown Predecessors.

parallel with the previous step a). The unknown vector shown contains a one–bit for all instructions whose mapping is not yet known. These are all loads, which did not send a NACK message and are not present in the local load queue. The logical AND of the predecessor mask and the unknown vector generates a vector that indicates the position of any unknown predecessors. If any of the bits in this vector is set, the early release is not possible.

To be able to identify unknown older store instructions, each valid entry of the load queue allocation vector contains the store sequence number of its closest store predecessor. A process similar to the one described in the last paragraph and illustrated in Figure 5.17 b) is performed for the store queue of the local cluster using the sequence number of the closest store predecessor of the candidate load.

The candidate load can only be released if the unknown predecessor vectors for both load and store queue contain only zeros.

The allocation vector is updated whenever an instruction is inserted or removed from the local queues. It directly reflects the contents of the local queues. If an entry is allocated the corresponding bit is set, if an entry is released, the bit is cleared.

The unknown vectors for both load and store queue are as well updated whenever the allocation vector is updated. However it takes the opposite action, if an entry is allocated the corresponding bit is cleared and if an entry is released a bit is set. The unknown vector is also updated when a NACK or remote commit is received. Upon a NACK the corresponding bit is cleared, upon a remote commit the corresponding bit is set.

NACK and remote commit messages are also used by the Conservative Deadlock Aware Entry Allocation mechanism described in Section 5.4 of this chapter. Since both mechanisms,

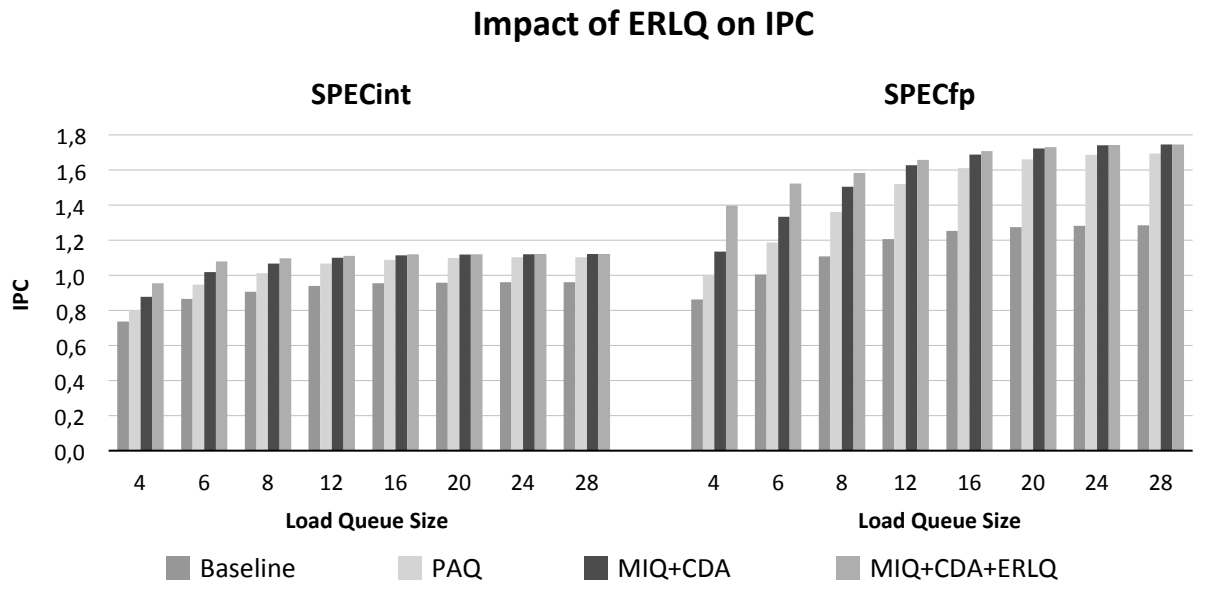

Figure 5.18: Impact of Early Release of Load Queue Entries (ERLQ) on Performance.

ERLQ and CDA, can be used in conjunction, analyze the same messages and use bit–vectors to manage internal information, they can share part of the implementation.

5.5.2 Early Release of Load Queue Entries: Evaluation

Figure 5.18 shows the impact of ERLQ on performance. To demonstrate the cumulative effect of the techniques presented in this chapter, we include configurations with an increasing number of features starting with a basic configuration with a pre–access queue, with a memory issue queue, then adding the CDA mechanism and finally adding the ERLQ.

For small queue sizes, there is a significant benefit of applying ERLQ. Middle queue sizes see some gain. Larger queue sizes (from 16 entries on upward for integer and 24 entries on upward for floating-point) see only a marginal improvement in performance, because the performance ultimately reaches a plateau and is no longer influenced by the queue size. However, using ERLQ this plateau can be reached with smaller queues. In the case of an integer workload, a configuration with only 6 queue entries achieves performance within 3.8% of the largest configuration (and with 8 entries within 2.2%). The floating-point benchmarks show even greater improvements in IPC and reach the performance plateau later. A configuration with 16 queue entries achieves floating-point performance within 2.3% of the largest configuration.

5.6 Quantitative Comparison to Previous Work

In this section we will compare our quantitative results to previous works, which distribute the disambiguation logic over several clusters, where each cluster contains a cache bank.

The most straightforward way to distribute the memory queues over several clusters are replicated queues where an entry is reserved for each instruction in each queue. This reservation can happen in-order while the instruction is being processed in the front-end of the processor. Only one entry in one cluster will be actually used by the instruction. Because the instructions in the queues must remain in program order and because a compacting queue is overly complex, the entries, which were reserved in other clusters for the same instruction, remain unused.

Yoaz et al. [Yoa99] propose to utilize a bank predictor for two tightly coupled memory pipelines. Zyuban and Kogge [Zyu01] extend this scheme to multiple more loosely coupled memory pipelines. This extended scheme reserves only one entry in one queue for an instruction if the bank predictor can deliver a confident prediction. Otherwise, the scheme falls back to replication and reserves entries in all queues. Hence, we will refer to this as predicted replication. It reserves queue entries in program order and allows the queue entries to stay physically ordered by age. This works fine as long as all confident predictions are correct. However, when a confident prediction turns out wrong, the affected instruction has to be inserted into another queue out of program order. We know of no way to insert an instruction out-of-order into an age–ordered queue, a pipeline flush may be the most realistic method to recover from this situation. To approximate the performance of this method we evaluate two variants of the scheme: an ideal variant, which can insert the instruction into the queue out-of-order, and a realistic variant, which has to recover from the situation with a pipeline flush.

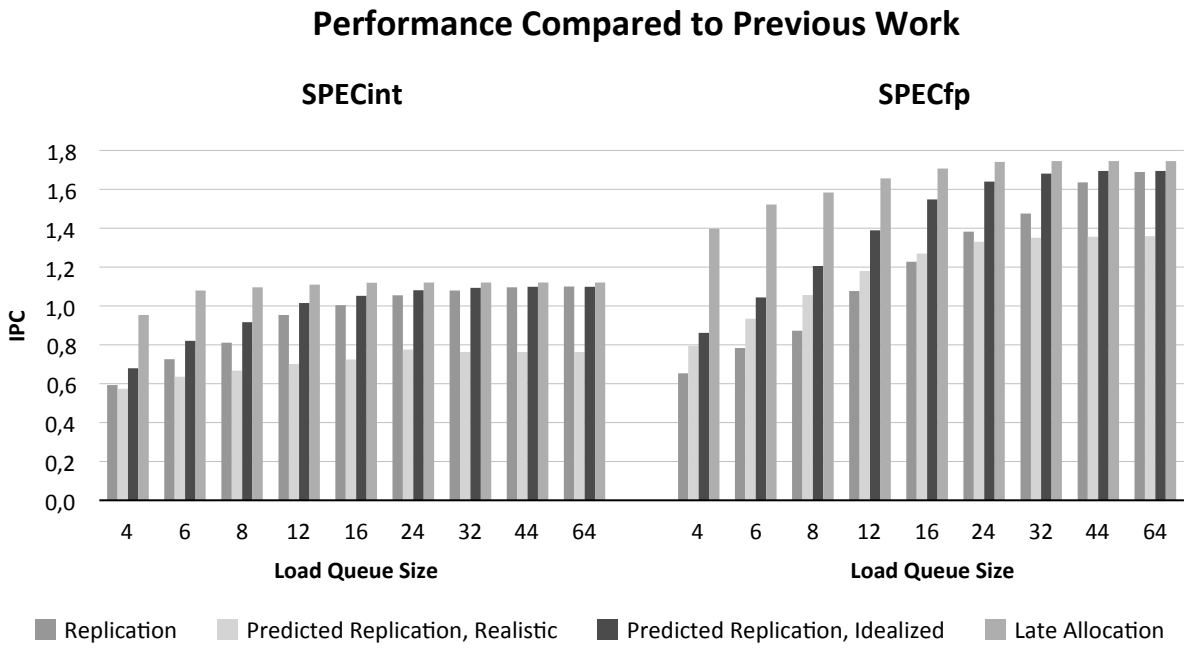

Figure 5.19: Performance Compared to Previous Work, IPC as a Function of Load Queue Size. Notice that the load queue size progresses exponentially rather than linearly as in previous figures.

In addition we show our own proposal, the distributed memory unit including unordered queues, late allocation, the memory issue queue, the conservative deadlock aware entry allocation and the early release of load queue entries. (The illustrations refer to this scheme as Late Allocation.)

Figure 5.19 shows the IPC of the different proposals as a function of load queue size. The first bar shows the performance of a scheme, which fully replicates the load queue. The second and third bar show the two variants of the predicted replication scheme. The performance of the realistic variant is severely constrained by the high number of pipeline flushes caused by a considerable number of incorrect confident bank predictions. The ideal variant shows much better performance, consistently higher than the fully replicated scheme. Our proposal (labeled Late Allocation) shows the best performance of all schemes and also reaches a slightly higher performance plateau. This is due to the impact of the memory issue queue and its issue policy giving priority to older instructions, which are more likely to be critical.

To illustrate the relation between performance and queue size better, we show the same data again in Figure 5.20. This time we plot the minimum load queue size required to attain a certain performance. Compared to the unordered queue the fully replicated scheme requires about three to four times as large queues to achieve the same performance. The ideal variant without pipeline flushes of the predicted replication scheme comes closer to our proposal and

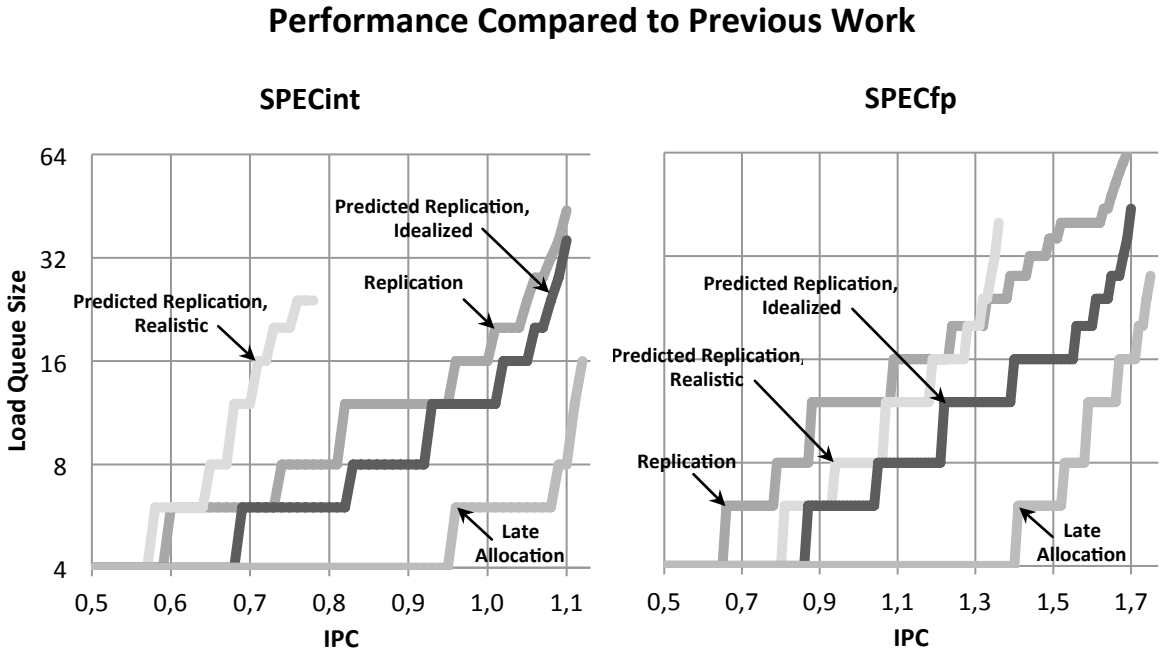

Figure 5.20: Performance Compared to Previous Work, Load Queue Size as a Function of IPC. This graph shows the same data as Figure 5.19 above. Notice the logarithmic scale of the y-axis. Colors of bars in Figure 5.19 are shown as lines of the same color in this figure. For each IPC the figure shows the minimum load queue size required. The end of the curve denotes the highest performance attainable by the scheme. The curves have the form of staircases because we didn't simulate all queue sizes (and because queue sizes are always discrete).

requires only about two to three times as large queues. The realistic variant doesn't reach comparable performance due to the high number of flush events.

We conclude this section with an estimation of performance and energy. In contrast to the estimations in Sections 1.2 and 3.6, for this evaluation we base our calculations on data from the microarchitectural simulation. We measure the performance in instructions per cycle. To estimate the energy of the memory unit we assign energy quantities to microarchitectural events and sum the quantities over the curse of the simulation. We count dynamic energies related to the memory unit including the data caches, the disambiguation logic, the bank predictors, and the inter-cluster communication. The individual energy quantities are calculated using CACTI. Appendix A contains the details of this experiment.

Figure 5.21 shows the results of this estimation. For SPECint as well as SPECfp the distributed configurations achieve higher performance and at the same time consume significantly less energy. The higher performance is due to the lower latency of load instructions. Both, the cache access latency and the average inter-cluster communication latency favor the distributed configuration. The centralized cache is four times bigger than the distributed cache and has a higher latency. Based on CACTI's cache model, we assume three cycles access latency for the centralized cache and two cycles for the distributed cache. The

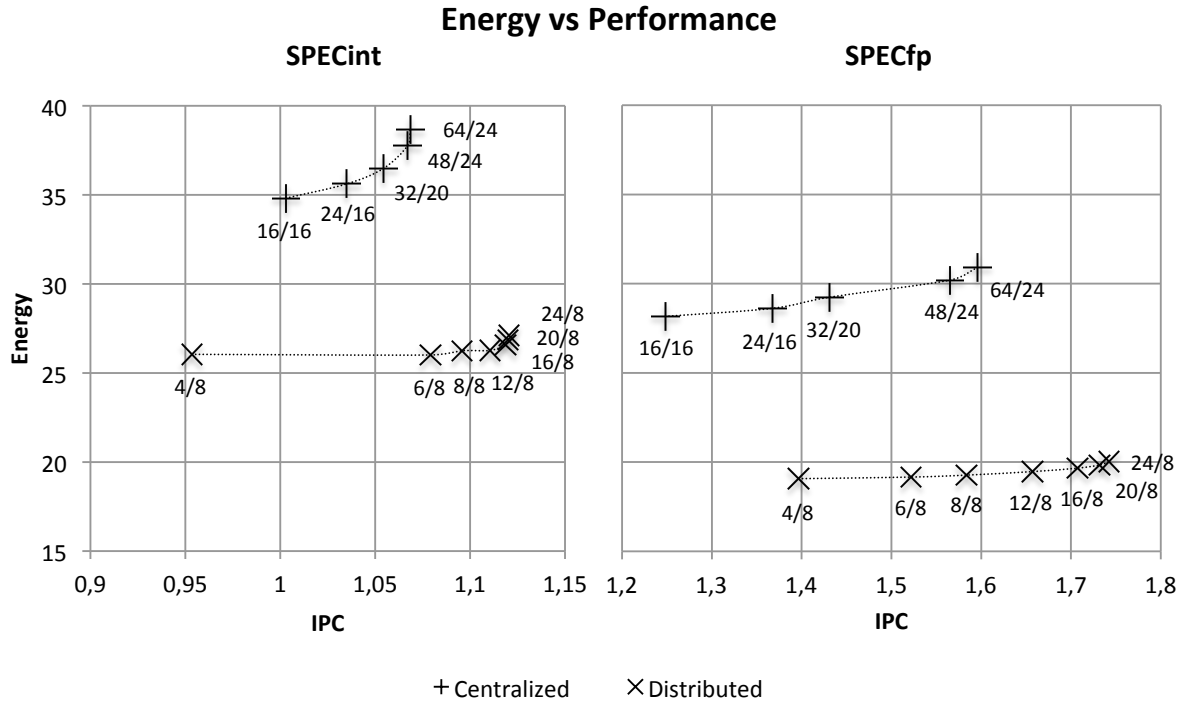

Figure 5.21: Energy and Performance for Centralized and Distributed Configurations. Only the energy usage of the memory unit and related inter-cluster communications are considered. The numbers near each data point indicate the sizes of the load/store queues. In case of the distributed configuration the sizes refer to one cluster (of four). The graph shows the results for a 32nm process, the scale on the left indicates picojoule per committed instruction. Graphs for 90nm, 65nm, and 45nm processes are very similar to the above and are not shown.

distributed cache exposes locality using the bank predictor while the centralized cache allows local cache accesses only for one of the four clusters.

As the sizes of load and store queues are increased, at some point, only the energy usage increases and the gains in performance become minuscule. This can be observed in the integer part of the figure. The floating-point benchmarks can make use of larger queues and the effect is not as visible because we did not include huge queue sizes to illustrate this point.

The increased queue size affects primarily the cost of disambiguation. Since the centralized configuration contains larger queues it is affected more by each size increase and the slope of the curve is higher than for the distributed configuration.

Figure 5.22 shows the energy components and the trend for shrinking process technologies. The relative advantage of the distributed configuration increases slightly with smaller process technologies while the distribution of energy usage between the components remains stable. Not included in Figure 5.21 and 5.22 is the increase in latency with smaller process technologies of the centralized cache relative to the distributed cache. For 90nm the centralized cache is 1.2 times slower than the distributed cache, for 32nm it is 1.7 times slower.

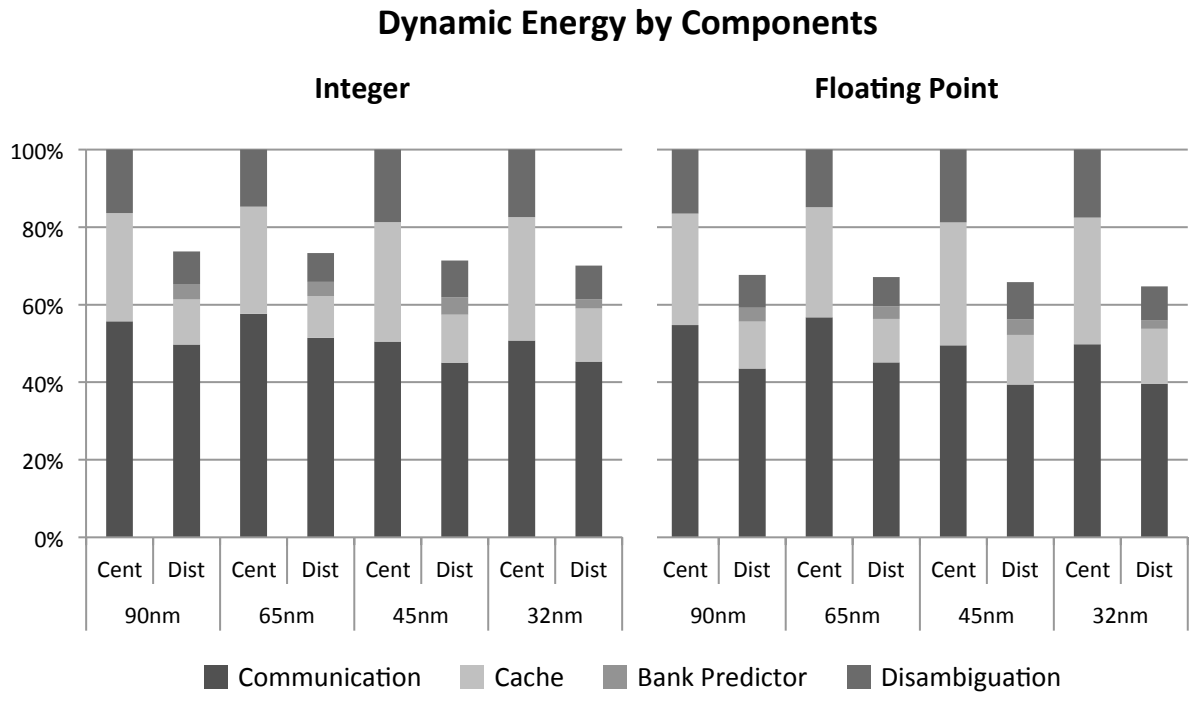

Figure 5.22: Dynamic Energy by Components for Centralized and Distributed Configurations. The centralized configuration shown includes a load queue of 64 entries and a store queue of 24 entries. The distributed configuration shown includes a load queue of 24 entries and a store queue of 8 entries, these sizes refer to one cluster (of four). The data is normalized for the centralized configuration and each process technology.

Comparing the energy usage of the different components, we observe that the distributed configuration uses less energy for each single component with the obvious exception of the bank predictor.

5.7 Conclusion

The techniques presented in this chapter improve the distributed memory unit introduced in the last chapter by increasing performance and decreasing the amount of speculatively executed instructions, thus improving the energy efficiency.

The flow control at dispatch and pre–access queue were able to remedy some shortcomings of our initial proposal. But the best results are achieved by a combination of the remaining three techniques—the memory issue queue, the conservative deadlock aware entry allocation and the early release of load queue entries. These three techniques form a symbiotic relationship. The memory issue queue allows an issue policy and thereby lays the basis for the conservative deadlock aware entry allocation. In addition, the mechanisms for entry allocation and release are based on the same concepts and can share part of their implementation.

CHAPTER 6

CONCLUSIONS

This chapter presents the main conclusions of the thesis and points out some open research areas, which may indicate directions for future work.

6.1 Conclusions

The first main proposal is a collection of cache bank predictors. We derive the bank predictors from well-known branch predictors as well as address/value predictors. The difference between these predictors and a bank predictor is primarily the size of the prediction. While branch predictors deliver just 1 bit and address/value predictors 32 or even 64 bits, our bank predictors have to deliver predictions of 3 bits to indicate the cache bank and an additional bit which indicates the confidence in the prediction.

We evaluate the bank predictors and compare them based on their accuracy and size. The most promising bank predictors are Gshare for small predictor sizes and a Tournament Predictor consisting of a Gshare and a Local predictor for larger predictors sizes. A Tournament Predictor of infinite size can reach more than 95% accuracy for SPECint benchmarks. A more reasonably sized Tournament Predictor of 3.3Kbytes still reaches 84% accuracy. Below this mark, a Gshare predictor is more attractive. At a size of 3.0Kbytes, it reaches 82% accuracy. While the predictors deliver satisfying accuracies, they are unable to

reduce the number of confident mispredictions to an acceptable level. This inability affected design decisions on our next proposal.

The second main proposal is the design of the distributed memory unit. Based on our experience with bank predictors we choose a design, which does not deploy confidence for bank predictions. Instead, we propose a novel steering scheme, which takes advantage of the bank predictor to reduce global communications. Because the distribution of memory instructions to clusters is based on the memory address, the precise mapping is unknown until late in the out-of-order stages of the pipeline. To solve this problem we allocate memory queue entries late and out-of-order. Allocating entries late has the additional benefit that it reduces the lifetime of each queue entry compared to standard early allocation. The out-of-order allocation turns out to be unsuited for ordinary age–ordered memory queues and we propose unordered memory queues to overcome this problem. Although our design attempts to limit speculation in order to increase its energy efficiency, to avoid a serious loss of performance we find it necessary to allow load instructions to issue before the addresses of all preceding store instructions are known.

The third main proposal is a collection of mechanisms to control the instruction flow. These mechanisms solve a problem, which is not present in conventional microarchitectures and which is related to the late allocation of memory queue entries. Traditionally entries in the memory queues are allocated in in-order pipeline stages before instructions enter the out-of-order core. If the memory queue runs out of free entries, the allocation stage stalls. Because we allocate queue entries out-of-order, this approach doesn't work, and our base architecture makes no attempt to control the instruction flow, which leads to frequent memory queue overflows and flush events. To overcome this problem we propose Dispatch Throttling, a scheme to control the instruction flow in the early in-order stages using three different heuristics as well as Pre–Access Queues a scheme to accommodate memory instructions near the memory queue to avoid overflows.

Our evaluations show that Dispatch Throttling reduces the number of flush events significantly—especially for smaller queues—but hardly affects performance. Throttling the dispatch stage reduces the number of instructions in the out-of-order core and limits parallelism. The gain by less frequent flush events is offset by this loss in parallelism. The Pre–Access Queue reduces flush events significantly and significantly improves performance for all benchmarks and queue sizes. The performance improves by 12% and 27% for SPECint and SPECfp respectively. The combination of both mechanisms slightly degrades performance and reduces the number of flush events further compared to the Pre–Access Queue alone.

The fourth main proposal is a mechanism to issue memory instructions using a Memory Issue Queue. This queue assumes the role of the Pre–Access Queue and extends its functionality. In contrast to the Pre–Access Queue, which acts as a FIFO, an issue queue can

apply a policy to select an instruction to issue. Selecting the oldest instruction in the queue improves performance because older instructions are more likely to be critical. To avoid frequent violations of memory dependencies the architecture includes a predictor, which indicates load instructions that should not execute speculatively. An issue queue allows these instructions to wait their turn in the issue queue while other instructions can continue to issue.

The fifth main proposal is a mechanism to intelligently issue instructions from the memory issue queue only if there is no danger of creating a deadlock in the memory queue. Deadlocks are a consequence of the out-of-order allocation of queue entries. They occur when younger instructions prevent older instructions from allocating a queue entry. The younger instructions do not release their queue entry until commit and the commit is delayed until all older instructions committed. This can lead to circular dependencies and prevent any forward progress—a deadlock. The proposed Conservative Deadlock Aware Entry Allocation delays the issue and allocation of queue entries until no deadlock can result.

Our evaluations show that this mechanism eliminates all deadlock events. This reduces the number of flush events especially for smaller memory queue sizes. Together with the Memory Issue Queue the Conservative Deadlock Aware Entry Allocation enables the use of small memory queues without causing a multitude of overflow and deadlock events. Both mechanisms improve performance and reduce the queue size required to achieve peak performance. For SPECint a load queue size of 12 entries (48 total) delivers the same performance as a queue with 24 entries (96 total) without the two mechanisms. For SPECfp a load queue size of 16 entries (64 total) delivers the same performance as a queue with 28 entries (96 total) without the two mechanisms.

The sixth main proposal is a mechanism to release load queue entries before the commit stage. Load queue entries are usually not freed until the commit stage. They take part in the disambiguation process and help to detect dependency violations with respect to older memory instructions. Once all older instructions have been verified, the entries no longer serve a useful purpose and can be freed. This mechanism may be applied to architectures with weak memory ordering like Alpha or PowerPC. Our evaluations show that this proposal increases the effective size of the load queue by 2 to 4 entries (8 to 16 total). This increase in effective size leads to a performance increase and a smaller queue size is required to reach peak performance.

This thesis demonstrated that a distributed memory unit is an attractive option for clustered microarchitectures. Compared to a centralized memory unit it uses significantly less energy and at the same time improves performance. This is possible because the distributed organization reduces the complexity of the disambiguation logic as well as the data cache. In addition, the communication between clusters can be optimized using a bank predictor, further decreasing energy consumption and latency. The proposals presented here identify and solve problems to which no solutions were previously described in the literature. We

demonstrated that these proposals lead to significant energy savings and performance increases.

6.2 Open Research Areas

In this thesis, we assumed that each backend is associated to exactly one memory unit. However, other scenarios are possible. E.g. for workloads (or workload phases) with little parallelism it may be advantageous to temporarily switch off one or more clusters to save energy, and in effect change the ratio of backends to memory units. Some physical layouts might also benefit from a different ratio as well as from a different topology of the interconnection network. Closely related to the topology are the communication protocols. Some optimizations are possible, considering both together. E.g. in a ring topology the broadcast messages required for some of our proposals can be combined with messages carrying load and store addresses from cluster to cluster. This would result in less total messages and save energy and possibly latency.

APPENDIX A

ENERGY ESTIMATION

This Appendix details the calculation used to generate Figure 5.21 in Chapter 5. To estimate the energy of the memory unit we assign energy quantities to microarchitectural events and sum the quantities over the curse of the simulation. We count dynamic energies related to the memory unit including the data caches, the disambiguation logic, the bank predictors, and the inter-cluster communication. The individual energy quantities are calculated using CACTI. For the sake of simplicity, we do not take into account memory issue queues, the translation look–aside buffer, cache misses, or external snoop events.

We use the same cache configurations as in the calculations in Section 1.2 of Chapter 1 and Section 3.6 of Chapter 3. The centralized configuration uses a cache of 64 Kbytes size, two ports, and 3 cycles access time. The distributed configuration uses four caches of 16 Kbytes size each, two ports each, and 2 cycles access time. The distributed configuration possesses a bank predictor, which is used to minimize communication between clusters.

Store queue size

The experiment requires a size of the store queue to calculate the energy cost of the disambiguation logic. Since the load queue size is our primary parameter for complexity, we set the store queue to a reasonable size depending on the load queue size. Overflows of the store queues are avoided either by memory queues in the case of the distributed memory unit

and by reserving the store queue entries in dispatch in case of the centralized memory unit. The primary effect of a small store queue is that not all stores can forward their data to load instructions. Missed store load forwards cause pipeline flushes, but can be avoided sometimes by the dependency predictor, see Section 4.2.3 in Chapter 4. We choose the store queue size for each configuration such that a missed store load forward happens about every 2000 instructions on average. For convenience, we select the nearest multiple of four as the store queue size. Choosing a comparatively low frequency of one flush every 2000 instructions assures that memory dependence flushes are much less frequent than branch prediction misses, which occur approximately every 200 instructions.

Disambiguation

When a load instruction issues it searches the store queue to find any memory dependencies. This search has to be fast so that the forwarding equals the speed of a cache access. Therefore, the search is not performed with the entire load address but a subset of 15 bits of the untranslated virtual address. Afterwards the search is performed again with the full address. This second search is not as time critical and serves to verify the correctness of the first search results. It is performed with the full physical address of 52 bits, which was not yet available at the time of issue. Load instructions also have to search the load queue to detect load-load dependencies. This search is not time critical and is performed with the physical address of 52 bits.

When a store instruction issues it searches the load queue to verify there are no violated memory dependencies. This search is not time critical and is performed with the physical address of 52 bits. Stores also search the store queue upon issue to update the no–hit–bit. (See Section 2.2.4 in Chapter 2 for an explanation of the no–hit–bit.) This search is performed with 15 bits of the virtual address to match the fast search for store load forwards.

Frequency of Microarchitectural Events

The following tables show the frequencies of some microarchitectural events relative to 100 committed instructions. The configurations are labeled to indicate a centralized or distributed memory unit, SPECint or SPECfp benchmarks, and the load queue size.

	Loads Decoded	Load Addresses Calculated	Store Addresses Calculated	Loads Issued	Stores Issued
Cent Int 16	41,039	33,275	12,007	30,017	10,901
Cent Int 24	41,079	33,221	11,984	30,600	10,884
Cent Int 32	41,108	33,167	11,947	30,879	10,893
Cent Int 48	41,044	33,153	11,825	31,047	10,925
Cent Int 64	41,125	33,126	11,950	31,085	10,921
Cent FP 16	28,638	26,428	8,394	25,243	8,152
Cent FP 24	28,707	26,465	8,381	25,375	8,138
Cent FP 32	28,761	26,497	8,389	25,475	8,149
Cent FP 48	28,703	26,473	8,361	25,471	8,122
Cent FP 64	28,736	26,488	8,383	25,491	8,143

	Loads Decoded	Load Addresses Calculated	Store Addresses Calculated	Loads Issued	Stores Issued
Dist Int 4	43,010	33,893	12,433	29,863	10,937
Dist Int 6	42,908	33,575	12,325	30,181	10,885
Dist Int 8	43,080	33,570	12,315	30,500	10,915
Dist Int 12	42,833	33,141	12,096	30,590	10,902
Dist Int 16	43,408	33,194	12,177	30,704	10,958
Dist Int 20	43,194	33,120	12,126	30,709	10,933
Dist Int 24	43,182	33,124	12,133	30,733	10,936
Dist FP 4	29,066	26,381	8,411	25,215	8,163
Dist FP 6	29,080	26,370	8,395	25,286	8,147
Dist FP 8	29,064	26,370	8,388	25,333	8,143
Dist FP 12	29,035	26,363	8,376	25,382	8,139
Dist FP 16	28,981	26,356	8,366	25,407	8,136
Dist FP 20	28,955	26,353	8,359	25,419	8,132
Dist FP 24	28,905	26,333	8,356	25,416	8,132

	Stores Committed	Load Adress Hops	Load Data Hops	Store Address Hops	Store Data Hops
Cent Int 16	9,677	32,785	29,734	11,917	11,917
Cent Int 24	9,677	32,706	30,313	11,910	11,910
Cent Int 32	9,677	32,701	30,528	11,904	11,904
Cent Int 48	9,677	32,649	30,693	11,889	11,889
Cent Int 64	9,677	32,665	30,724	11,764	11,764
Cent FP 16	7,997	26,250	25,011	8,420	8,420
Cent FP 24	7,997	26,256	25,137	8,436	8,436
Cent FP 32	7,998	26,274	25,234	8,395	8,395
Cent FP 48	7,998	26,375	25,222	8,383	8,383
Cent FP 64	7,999	26,282	25,246	8,360	8,360
Dist Int 4	9,677	33,906	9,617	11,652	11,652
Dist Int 6	9,677	33,365	9,632	11,523	11,523
Dist Int 8	9,677	33,587	9,643	11,544	11,544
Dist Int 12	9,677	33,225	9,601	11,342	11,342
Dist Int 16	9,677	32,887	9,884	11,452	11,452
Dist Int 20	9,677	33,113	9,924	11,457	11,457
Dist Int 24	9,677	33,135	9,922	11,462	11,462
Dist FP 4	7,998	27,067	3,815	7,268	7,268
Dist FP 6	7,998	27,068	3,812	7,240	7,240
Dist FP 8	7,999	27,018	3,831	7,275	7,275
Dist FP 12	7,998	27,004	3,883	7,270	7,270
Dist FP 16	7,998	27,046	3,915	7,255	7,255
Dist FP 20	7,997	26,982	3,901	7,263	7,263
Dist FP 24	7,997	26,989	3,901	7,264	7,264

The following table shows energy quantities, which we will later assign to microarchitectural events. The quantities are given for four different process technologies in picojoule. These values have been obtained with CACTI 6.5. [Mur09]

	Bank Predictor Read	Bank Predictor Write	Unit Distance	Cache Read Centralized	Cache Write Centralized	Cache Read Distributed	Cache Write Distributed
90nm	15,877	11,759	1,481	187,627	178,766	78,910	75,192
65nm	8,344	5,764	0,861	104,839	96,236	41,203	38,358
45nm	5,036	3,451	0,365	57,436	50,172	22,800	21,571
32nm	1,430	0,936	0,193	31,497	25,927	12,834	13,843

The following table shows the cost searching the load and store queues. A fast search refers to a CAM search, which is only 15 bits wide, all other searches are 52 bits wide. All energies are in picojoule. These values have been obtained with CACTI 6.5. [Mur09]

	90nm	65nm	45nm	32nm
STQ Fast Search 8	6,396	3,146	1,954	0,931
STQ Fast Search 16	9,153	4,528	2,819	1,351
STQ Fast Search 20	10,531	5,220	3,252	1,562
STQ Fast Search 24	11,910	5,911	3,684	1,772
STQ Slow Search 8	19,566	9,677	5,961	2,854
STQ Slow Search 16	26,579	13,273	8,162	3,947
STQ Slow Search 20	30,085	15,071	9,262	4,494
STQ Slow Search 24	33,592	16,868	10,363	5,040
LDQ Search 4	16,059	7,879	4,861	2,307
LDQ Search 6	17,812	8,778	5,411	2,580
LDQ Search 8	19,566	9,677	5,961	2,854
LDQ Search 12	23,072	11,475	7,061	3,400
LDQ Search 16	26,579	13,273	8,162	3,947
LDQ Search 20	30,085	15,071	9,262	4,494
LDQ Search 24	33,592	16,868	10,363	5,040
LDQ Search 32	40,605	20,464	12,563	6,134
LDQ Search 48	54,631	27,656	16,965	8,320
LDQ Search 64	68,657	34,847	21,367	10,507

The following table shows the energy estimation for the bank predictor. We assign the energy of a predictor read to each decoded load instruction and the energy of a predictor write to each calculated load address. The table shows the average energy in picojoule for each committed instruction.

	Integer				Floating-Point			
LDQ/STQ	**90nm**	**65nm**	**45nm**	**32nm**	**90nm**	**65nm**	**45nm**	**32nm**
Dist 4 / 8	10,814	5,542	3,336	0,932	7,717	3,946	2,374	0,663
Dist 6 / 8	10,760	5,516	3,319	0,928	7,718	3,946	2,374	0,663
Dist 8 / 8	10,787	5,530	3,328	0,930	7,715	3,945	2,374	0,663
Dist 12 / 8	10,697	5,484	3,301	0,923	7,710	3,942	2,372	0,662
Dist 16 / 8	10,795	5,535	3,331	0,932	7,700	3,937	2,369	0,661
Dist 20 / 8	10,752	5,513	3,318	0,928	7,696	3,935	2,368	0,661
Dist 24 / 8	10,751	5,512	3,318	0,928	7,686	3,930	2,364	0,660

The following table shows the energy estimation for the cache accesses. We assign the energy of a cache read to each issued load and the energy of a cache write to each committed store. The table shows the average energy in picojoule for each committed instruction.

	Integer				Floating-Point			
LDQ/STQ	**90nm**	**65nm**	**45nm**	**32nm**	**90nm**	**65nm**	**45nm**	**32nm**
Cent 16 / 16	73,620	40,783	22,096	11,964	61,659	34,160	18,511	10,024
Cent 24 / 16	74,714	41,394	22,431	12,147	61,906	34,299	18,587	10,066
Cent 32 / 20	75,238	41,687	22,591	12,235	62,095	34,404	18,645	10,098
Cent 48 / 24	75,553	41,863	22,688	12,288	62,088	34,400	18,642	10,096
Cent 64 / 24	75,625	41,903	22,710	12,300	62,127	34,422	18,654	10,103
Dist 4 / 8	30,841	16,016	8,896	5,172	25,910	13,457	7,474	4,343
Dist 6 / 8	31,092	16,147	8,969	5,213	25,966	13,486	7,490	4,352
Dist 8 / 8	31,344	16,279	9,042	5,254	26,005	13,506	7,501	4,359
Dist 12 / 8	31,415	16,316	9,062	5,266	26,043	13,526	7,512	4,365
Dist 16 / 8	31,505	16,363	9,088	5,280	26,062	13,536	7,518	4,368
Dist 20 / 8	31,509	16,365	9,089	5,281	26,071	13,541	7,521	4,369
Dist 24 / 8	31,528	16,375	9,095	5,284	26,069	13,540	7,520	4,369

The following table shows the energy estimation for the disambiguation. To each issued load instruction we assign the energy of a fast and a slow store queue search as well as a load queue search. To each issued store instruction we assign the energy of a fast store queue search and a load queue search.

	Integer				Floating-Point			
LDQ/STQ	**90nm**	**65nm**	**45nm**	**32nm**	**90nm**	**65nm**	**45nm**	**32nm**
Cent 16 / 16	22,599	11,268	6,943	3,353	18,642	9,295	5,727	2,766
Cent 24 / 16	25,865	12,938	7,966	3,859	21,069	10,538	6,489	3,143
Cent 32 / 20	30,651	15,382	9,466	4,602	24,858	12,475	7,677	3,732
Cent 48 / 24	38,358	19,326	11,884	5,801	30,909	15,572	9,576	4,674
Cent 64 / 24	44,286	22,365	13,744	6,725	35,661	18,008	11,067	5,415
Dist 4 / 8	15,004	7,388	4,561	2,173	12,428	6,120	3,778	1,800
Dist 6 / 8	15,846	7,817	4,824	2,303	13,041	6,433	3,970	1,895
Dist 8 / 8	16,719	8,262	5,096	2,438	13,647	6,744	4,160	1,990
Dist 12 / 8	18,212	9,026	5,564	2,670	14,844	7,357	4,535	2,176
Dist 16 / 8	19,745	9,811	6,045	2,908	16,032	7,966	4,908	2,361
Dist 20 / 8	21,200	10,557	6,501	3,135	17,213	8,571	5,278	2,545
Dist 24 / 8	22,676	11,314	6,964	3,365	18,387	9,174	5,647	2,728

The following table shows the energy estimation for the communication. To each hop of load/store data/address, we assign the energy for a unit distance multiplied by 78 bits (to account for 64 bits of address/data and 14 bits of overhead like instruction type, destination cluster, sequence number, access size, etc.) and multiplied by a factor of 1.5. The connection between the left and right cluster is physically three times larger than the connections between neighboring clusters. (Compare Figure 1.1 in Chapter 1.) We assume that messages are distributed equally to all connections. The average physical distance therefore is 1.5 unit distances.

	Integer				Floating-Point			
LDQ/STQ	**90nm**	**65nm**	**45nm**	**32nm**	**90nm**	**65nm**	**45nm**	**32nm**
Cent 16 / 16	149,654	86,943	36,880	19,493	118,021	68,566	29,084	15,373
Cent 24 / 16	150,496	87,433	37,087	19,603	118,309	68,733	29,155	15,410
Cent 32 / 20	150,839	87,632	37,172	19,647	118,362	68,764	29,168	15,417

	Integer				Floating-Point			
Cent 48 / 24	150,986	87,717	37,208	19,666	118,476	68,830	29,196	15,432
Cent 64 / 24	150,631	87,511	37,121	19,620	118,276	68,714	29,147	15,406
Dist 4 / 8	136,400	79,243	33,614	17,766	94,172	54,710	23,207	12,266
Dist 6 / 8	134,852	78,344	33,232	17,565	94,060	54,645	23,179	12,252
Dist 8 / 8	135,322	78,617	33,348	17,626	94,124	54,682	23,195	12,260
Dist 12 / 8	133,636	77,638	32,932	17,406	94,166	54,707	23,206	12,265
Dist 16 / 8	133,978	77,836	33,017	17,451	94,234	54,746	23,222	12,274
Dist 20 / 8	134,404	78,084	33,122	17,506	94,120	54,680	23,194	12,259
Dist 24 / 8	134,460	78,116	33,135	17,514	94,125	54,683	23,196	12,260

The following table shows the resulting total energy estimation for each configuration. The unit is picojoule per committed instruction.

	Integer				Floating-Point			
LDQ/STQ	**90nm**	**65nm**	**45nm**	**32nm**	**90nm**	**65nm**	**45nm**	**32nm**
Cent 16 / 16	245,872	138,994	65,919	34,809	198,321	112,021	53,323	28,162
Cent 24 / 16	251,076	141,765	67,484	35,609	201,284	113,570	54,230	28,619
Cent 32 / 20	256,728	144,701	69,230	36,484	205,316	115,644	55,490	29,247
Cent 48 / 24	264,896	148,905	71,780	37,755	211,472	118,803	57,415	30,202
Cent 64 / 24	270,541	151,778	73,574	38,645	216,064	121,145	58,868	30,923
Dist 4 / 8	193,060	108,190	50,406	26,044	140,227	78,233	36,833	19,072
Dist 6 / 8	192,551	107,824	50,344	26,009	140,784	78,511	37,014	19,162
Dist 8 / 8	194,173	108,688	50,814	26,248	141,491	78,878	37,230	19,271
Dist 12 / 8	193,961	108,465	50,860	26,265	142,763	79,533	37,625	19,468
Dist 16 / 8	196,023	109,546	51,481	26,571	144,028	80,186	38,017	19,664
Dist 20 / 8	197,865	110,519	52,030	26,850	145,100	80,728	38,361	19,835
Dist 24 / 8	199,414	111,317	52,512	27,090	146,267	81,326	38,727	20,017

We obtained the performance data from our microarchitectural simulations and show the data in instructions per cycle. The two numbers in the name of each configuration indicate the size of the load / store queue respectively. Results are for SPECint and SPECfp.

	Integer	Floating-Point
Cent 16 / 16	1,003	1,248
Cent 24 / 16	1,035	1,367
Cent 32 / 20	1,054	1,432
Cent 48 / 24	1,067	1,565
Cent 64 / 24	1,068	1,596
Dist 4 / 8	0,953	1,397
Dist 6 / 8	1,079	1,522
Dist 8 / 8	1,096	1,583
Dist 12 / 8	1,111	1,657
Dist 16 / 8	1,119	1,707
Dist 20 / 8	1,120	1,732
Dist 24 / 8	1,120	1,742

BIBLIOGRAPHY

[Abr97] J.M. Abramson, H. Akkary, A.F. Glew, G.J. Hinton, K.G. Konigsfeld, P.D. Madland, “Method and apparatus for performing load operations in a computer system”, United States Patent no. 5694574, 1997.

[Aga00] V. Agarwal, M. Hrishikesh, S. Keckler, and D. Burger, “Clock rate versus IPC: the end of the road for conventional microarchitectures,” International Symposium on Computer Architecture, 2000, ISCA–27, pp. 248–259.

[Agg05] A. Aggarwal, “Reducing latencies of pipelined cache accesses through set prediction,” International Conference on Supercomputing, 2005, ICS–19, pp. 2–11.

[Alpha] “Alpha Architecture Reference Manual”, Compaq Computer Corporation, Version 4, Order Number EC–QD2KC–TE, 1998.

[AEV6] “Alpha 21264 Microprocessor Hardware Reference Manual,” Compaq Computer Corporation, Order Number: EC–RJRZA–TE, Version 1, 1999.

[AMD02] “AMD Athlon Processor x86 Code Optimization Guide”, Appendix A “Microarchitecture” and Appendix B “Pipeline and Execution Unit Resources Overview,” AMD Publication no. 22007, 2002, pp 203–228.

[Amd67] G.M. Amdahl, "Validity of the Single–Processor Approach to Achieving Large–Scale Computing Capabilities," American Federation of Information Processing Societies Conference, AFIPS Press, 1967, pp. 483–485.

[And67] D.W. Anderson, F.J. Sparacio, and R.M. Tomasulo, “The IBM System/360 Model 91: Machine Philosophy and Instruction–Handling,” IBM Journal of Research and Development, vol. 11, no. 1, 1967, pp. 8–24.

[Aus02] T. Austin, E. Larson, D. Ernst, “SimpleScalar: an infrastructure for computer system modeling,” IEEE Computer, Volume 35, Issue 2, 2002, pp. 59–67.

[Bal02] R. Balasubramonian, S. Dwarkadas, and D.H. Albonesi, “Microarchitectural Trade-offs in the Design of a Scalable Clustered Microprocessor,” University of Rochester, URCS Technical Report 771, 2002.

[Bal03] R. Balasubramonian, S. Dwarkadas, and D.H. Albonesi, “Dynamically managing the communication–parallelism trade-off in future clustered processors,” International Symposium on Computer Architecture, 2003, ISCA–30, pp. 275–287.

[Bal04] R. Balasubramonian, "Cluster prefetch: tolerating on-chip wire delays in clustered microarchitectures," International Conference on Supercomputing, 2004, ICS–18, pp. 326–335.

[Bal05] S. Balakrishnan, Ravi Rajwar, M. Upton, and K. Lai, "The impact of performance asymmetry in emerging multicore architectures," International Symposium on Computer Architecture, 2005, ISCA–32, pp. 506–517.

[Ban00] A. Baniasadi and A. Moshovos, "Instruction distribution heuristics for quad–cluster, dynamically–scheduled, superscalar processors," International Symposium on Microarchitecture, 2000, MICRO–33, pp. 337–347.

[Bed03] M. Bedford Taylor, W. Lee, S. Amarasinghe, and A. Agarwal, "Scalar operand networks: on-chip interconnect for ILP in partitioned architectures," International Symposium on High–Performance Computer Architecture, 2003. HPCA–9, pp. 341–353.

[Bek99] M. Bekerman, S. Jourdan, R. Ronen, G. Kirshenboim, L. Rappoport, A. Yoaz, and U. Weiser, "Correlated load–address predictors," International Symposium on Computer Architecture, 1999, ISCA–26, pp. 54–63.

[Bek00] M. Bekeman, A. Yoaz, F. Gabbay, S. Jourdan, M. Kalaev, and R. Ronen, "Early load address resolution via register tracking," International Symposium on Computer Architecture, 2000, ISCA–27, pp. 306–315.

[Ber03] K. Bernstein, "Microarchitecture on the MOSFET diet," International Symposium on Microarchitecture, 2003. MICRO–36,p. 3.

[Bha03] R. Bhargava and L. John, "Improving dynamic cluster assignment for clustered trace cache processors," International Symposium on Computer Architecture, 2003, ISCA–30, pp. 264–274.

[Bie05] S. Bieschewski, J. Parcerisa, and A. Gonzalez, "Memory bank predictors," International Conference on Computer Design, 2005, ICCD–23, pp. 666–668.

[Bie07] S. Bieschewski, J.-M. Parcerisa, A. González, "A Fully–Distributed First Level Memory Architecture," Technical Report UPC–DAC–RR–ARCO–2007–3, Universitat Politècnica de Catalunya, 2007.

[Boh95] M.T. Bohr, "Interconnect scaling—the real limiter to high performance ULSI," International Electron Devices Meeting, 1995, IEDM, pp. 241–244.

[Bol67] L.J. Boland, G.D. Granito, A.U.Marcotte, B.U. Messina, and J.W. Smith, "The IBM System/360 Model 91: Storage System," IBM Journal of Research and Development, vol. 11, no. 1, 1967, pp. 54–68.

[Buy02] A. Buyuktosunoglu, D.H. Albonesi, P. Bose, P.W. Cook, S.E. Schuster, "Tradeoffs in Power–Efficient Issue Queue Design," Low Power Electronics and Design, ISLPED, pp. 184–189.

[Cai04] H. Cain and M. Lipasti. "Memory Ordering: A Value Based Approach," International Symposium on Computer Architecture, ISCA–31, 2004, pp. 90–101.

[Can00] R. Canal, J. Parcerisa, and A. Gonzalez, "Dynamic cluster assignment mechanisms," International Symposium on High–Performance Computer Architecture, 2000, HPCA–6, pp. 133–142.

[Cas06] F. Castro, L. Pinuel, D. Chaver, M. Prieto, M. Huang, and F. Tirado, "DMDC: Delayed Memory Dependence Checking through Age–Based Filtering," International Symposium on Microarchitecture, 2006, MICRO–38, pp. 297–308.

[Cez07] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas, "BulkSC: bulk enforcement of sequential consistency," , International Symposium on Computer Architecture, 2007, ISCA–34, pp. 278–289.

[Che04] L. Chen, D.H. Albonesi, and S. Dropsho, "Dynamically Matching ILP Characteristics Via a Heterogeneous Clustered Microarchitecture," IBM Watson Conference on the Interaction Between Architectures, Circuits, and Compilers, 2004, pp. 136–143.

[Chi03] Z. Chishti, M. Powell, and T. Vijaykumar, "Distance associativity for high–performance energy–efficient non-uniform cache architectures," International Symposium on Microarchitecture, 2003, MICRO–36, pp. 55–66.

[Chr98] G. Chrysos and J. Emer, "Memory dependence prediction using store sets," International Symposium on Computer Architecture, 1998, ISCA–25, pp. 142–153.

[Dav01] J. Davis, R. Venkatesan, A. Kaloyeros, M. Beylansky, S. Souri, K. Banerjee, K. Saraswat, A. Rahman, R. Reif, and J. Meindl, "Interconnect limits on gigascale integration (GSI) in the 21st century," Proceedings of the IEEE, vol. 89, iss. 3, 2001, pp. 305–324.

[Dav05] J. Davis, J. Laudon, and K. Olukotun, “Maximizing CMP throughput with mediocre cores,” International Conference on Parallel Architectures and Compilation Techniques, 2005, PACT–14, pp. 51–62.

[Den74] R. H. Dennard, F. H. Gaensslen, Hwa-Nien. Yu, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of ion–implanted MOSFETS with very small physical dimensions," IEEE Journal of Solid State Circuits, Vol. SC–9, 1974, pp. 256–268.

[Die98] K. Diefendorff, “K7 Challenges Intel—New AMD Processor Could Beat Intel’s Katmai,” Microprocessor Report, vol. 12, no. 14, 1998, pp 1–7.

[Dro04] S. Dropsho, G. Semeraro, D. Albonesi, G. Magklis, and M. Scott, “Dynamically Trading Frequency for Complexity in a GALS Microprocessor,” International Symposium on Microarchitecture, 2004, MICRO–37, pp. 157–168.

[Eic93] R. J. Eickemeyer and S. Vassiliadis, “A load instruction unit for pipelined processors,” IBM Journal of Research and Development, vol. 37, 1993, pp. 547–564.

[ESA390] “Enterprise Systems Architecture/390 Principles of Operation”, International Business Machines, 9th edition, Document Number SA22–7201–08, 2003.

[Fan06] C. Fang, S. Carr, S. Önder, and Z. Wang, “Feedback–directed memory disambiguation through store distance analysis,” International Conference on Supercomputing, 2006, ICS–20, pp. 278–287.

[Far97] K. Farkas, P. Chow, N. Jouppi, and Z. Vranesic, “The multicluster architecture: reducing cycle time through partitioning,” International Symposium on Microarchitecture, 1997, MICRO–30, pp. 149–159.

[Fis83] J.A. Fisher, “Very Long Instruction Word architectures and the ELI–512”, International Symposium on Computer Architecture, 1983, ISCA–10, pp 140–150.

[Fra96] M. Franklin and G.S. Sohi, “ARB: a hardware mechanism for dynamic reordering of memory references,” IEEE Transactions on Computers, vol. 45, iss. 5, 1996, pp. 552–571.

[Fre05] B. Frey, “PowerPC Architecture Book”, International Business Machines, Version 2.02, 2005.

[Ful11] S. H. Fuller and L. I. Millett (Editors), “The Future of Computing Performance: Game Over or Next Level?” The National Academy Press, 2011.

[Gan05] A. Gandhi, H. Akkary, R. Rajwar, S. Srinivasasn, and K. Lai, "Scalable load and store processing in latency tolerant processors," ACM SIGARCH Computer Architecture News, vol. 33, iss. 2, 2005, pp. 446–457.

[Gar06] A. Garg, M.W. Rashid, and M. Huang, "Slackened Memory Dependence Enforcement: Combining Opportunistic Forwarding with Decoupled Verification," International Symposium on Computer Architecture, 2006, ISCA–33, pp. 142–154.

[Gha95] K. Gharachorloo, "Memory Consistency Models for Shared–Memory Multiprocessors," Technical Report: CSL–TR–95–685, Stanford University, 1995.

[Gib10] D. Gibson, D. A. Wood, "Forwardflow: A Scalable Core for Power–Constrained CMPs," International Symposium on Computer Architecture, 2010, ISCA–37, pp. 14–25.

[Goe01] B. Goeman, H. Vandierendonck, and K. De Bosschere, "Differential FCM: Increasing Value Prediction Accuracy by Improving Table Usage Efficiency," International Symposium on High–Performance Computer Architecture, 2001, HPCA–7, pp. 207–216.

[Gun07] E. Gunadi and M. Lipasti, "A position-insensitive finished store buffer," International Conference on Computer Design, 2007, ICCD-25, pp. 105–112.

[Gwe95] L. Gwennap, "Intel's P6 Uses Decoupled Superscalar Design," Microprocessor Report, vol. 9, no. 2, MicroDesign Resources, 1995.

[Gwe96] L. Gwennap, "Digital 21264 Sets New Standard," Microprocessor Report, vol. 10, no. 14, MicroDesign Resources, 1996.

[Har03] A. Hartstein and T. Puzak, "Optimum power/performance pipeline depth," International Symposium on Microarchitecture, 2003. MICRO–36, pp. 117–125.

[Hil08] M. D. Hill, M. R. Marty, "Amdahl's Law in the Multicore Era," IEEE Computer July 2008, pp. 33–38.

[Hu06] S. Hu, I. Kim, M. Lipasti, and J. Smith, "An approach for implementing efficient superscalar CISC processors," International Symposium on Computer Architecture, 2006, ISCA–29, pp. 41–52.

[Hua94] A.S. Huang, G. Slavenburg, and J.P. Shen, "Speculative disambiguation: a compilation technique for dynamic memory disambiguation," International Symposium on Computer Architecture, 1994, ISCA–21, pp. 200–210.

[Hua06] R. Huang, A. Garg, and M. Huang, “Software–hardware cooperative memory disambiguation,” International Symposium on High–Performance Computer Architecture, 2006, HPCA–12, pp. 244–253.

[Hug02] W.A. Hughes, and J.S. Roberts, “Load/store unit employing last–in–buffer indication for rapid load–hit–store”, United States Patent no. 6393536, 2002.

[Huh05] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S.W. Keckler, “A NUCA substrate for flexible CMP cache sharing,” International Conference on Supercomputing, 2005, ICS–19, pp. 31–40.

[Hun95] D. Hunt, “Advanced Performance Features of the 64–bit PA–8000”, COMPCON 95 Technologies for the Information Superhighway, 1995, COMPCON–95, pp. 123–128.

[IA32] “Intel® 64 and IA–32 Architectures Software Developer's Manual, System Programming Guide,” Intel Corporation, Order Number 325384, Version 039, 2011.

[IA64M] “A Formal Specification of Intel® Itanium® Processor Family Memory Ordering,” Intel Corporation, Application Note, Order Number 251429, Version 001, 2002.

[Ipe07] E. Ipek, M. Kirman, N. Kirman, and J.F. Martinez, “Core fusion: accommodating software diversity in chip multiprocessors,” International Symposium on Computer Architecture, 2007, ISCA–34, pp. 186–197.

[Iri05] H. Irie, N. Hattori, M. Takada, N. Hatta, T. Toyoshima, and S. Sakai, “Steering and forwarding techniques for reducing memory communication on a clustered microarchitecture,” International Workshop on Innovative Architecture for Future Generation High–Performance Processors and Systems, 2005, IWIA–05, pp. 13–18.

[Jal05] J. Jaleel and B. Jacob, “Using virtual load/store queues (VLSQs) to reduce the negative effects of reordered memory instructions,” International Symposium on High–Performance Computer Architecture, 2005, HPCA–11, pp. 191–200.

[Kes98] R.E. Kessler, E.J. McLellan, and D.A. Webb, “The Alpha 21264 Microprocessor Architecture,” International Conference on Computer Design, 1998, ICCD–16, pp. 90–95.

[Kim02] H.S. Kim and J. Smith, "An instruction set and microarchitecture for instruction level distributed processing," International Symposium on Computer Architecture, 2002, ISCA–29, pp. 71–81.

[Kim04] I. Kim and M. Lipasti, "Understanding scheduling replay schemes," International Symposium on High Performance Computer Architecture, 2004, HPCA–10, pp. 198–209.

[Kir05] N. Kirman, M. Kirman, M. Chaudhuri, and J. Martinez, "Checkpointed early load retirement," International Symposium on High–Performance Computer Architecture, 2005, HPCA–11, pp. 16–27.

[Kur03] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen, "Single–ISA heterogeneous multi–core architectures: the potential for processor power reduction," International Symposium on Microarchitecture, 2003, MICRO–36, pp. 81–92.

[Kum04a] R. Kumar, D. Tullsen, P. Ranganathan, N. Jouppi, and K. Farkas, "Single–ISA heterogeneous multi–core architectures for multithreaded workload performance," International Symposium on Computer Architecture, 2004, ISCA–31, pp. 64–75.

[Kum04b] R. Kumar, N. Jouppi, and D. Tullsen, "Conjoined–Core Chip Multiprocessing," International Symposium on Microarchitecture, 2004, MICRO–37, pp. 195–206.

[Kum05] R. Kumar, V. Zyuban, and D. Tullsen, "Interconnections in multi–core architectures: understanding mechanisms, overheads and scaling," International Symposium on Computer Architecture, 2005, ISCA–32, pp. 408–419.

[Kum06] R. Kumar, D.M. Tullsen, and N.P. Jouppi, "Core architecture optimization for heterogeneous chip multiprocessors," International Conference on Parallel Architectures and Compilation Techniques, 2006, PACT–16, pp. 23–32.

[Lat04] F. Latorre, J. González, and A. González, "Back-end assignment schemes for clustered multithreaded processors," International Conference on Supercomputing, 2004, ICS–18, pp. 316–325.

[Lep00a] K. Lepak and M. Lipasti, "On the value locality of store instructions," International Symposium on Computer Architecture, 2000, ISCA–27, pp. 182–191.

[Lep00b] K. Lepak and M. Lipasti, "Silent stores for free," International Symposium on Microarchitecture, 2000, MICRO–33, pp. 22–31.

[Lia07] X. Liang, K. Turgay, and D. Brooks, "Architectural Power Models for SRAM and CAM Structures Based on Hybrid Analytical/Empirical Techniques," International Conference on Computer Aided Design, 2007, ICCAD–07, pp. 824–830.

[Loh02] G. H. Loh, R. Sami, and D. H. Friendly, "Memory Bypassing: Not Worth the Effort," Workshop on Duplicating, Deconstructing, and Debunking, 2002, WDDD–1, pp. 71–80.

[McF93] S. McFarling, "Combining Branch Predictors,", Technical Report TN–36, Western Research Lab, 1993.

[Mei02] J.D. Meindl, J.A. Davis, P. Zarkesh-Ha, C.S. Patel, K.P. Martin, and P.A. Kohl, "Interconnect opportunities for gigascale integration," IBM Journal of Research and Development, vol. 46, iss. 2.3, 2002, pp. 245–263.

[Mic97] P. Michaud, A. Seznec, and R. Uhlig. "Trading conflict and capacity aliasing in conditional branch predictors," International Symposium on Computer Architecture, 1997, ISCA–24, pp. 292–303.

[Mos97a] A. Moshovos, S. Breach, T. Vijaykumar, and G. Sohi, "Dynamic Speculation And Synchronization Of Data Dependence," International Symposium on Computer Architecture, 1997, ISCA–24, pp. 181–193.

[Mos97b] A. Moshovos and G. Sohi, "Streamlining inter–operation memory communication via data dependence prediction," International Symposium on Microarchitecture, 1997, MICRO–30, pp. 235–245.

[Mos99] A. Moshovos and G. Sohi, "Read–after–read memory dependence prediction," International Symposium on Microarchitecture, 1999, MICRO–32, pp. 177–185.

[Mos00] A. Moshovos and G. Sohi, "Memory dependence speculation tradeoffs in centralized, continuous–window superscalar processors," International Symposium on High–Performance Computer Architecture, 2000, HPCA–6, pp. 301–312.

[Mur07] N. Muralimanohar and R. Balasubramonian, "Interconnect design considerations for large NUCA caches," International Symposium on Computer Architecture, ISCA–34, 2007, pp. 369–380.

[Mur09] N. Muralimanohar, R. Balasubramonian, N. P. Jouppi, "CACTI 6.0: A Tool to Model Large Caches," Technical Report HPL–2009–85, HP Laboratories, 2009.

[Nag01] R. Nagarajan, K. Sankaralingam, D. Burger, and S. Keckler, "A design space evaluation of grid processor architectures," International Symposium on Microarchitecture, 2001, MICRO–34, pp. 40–51.

[Nag04] R. Nagarajan, S. Kushwaha, D. Burger, K. McKinley, C. Lin, and S. Keckler, "Static placement, dynamic issue (SPDI) scheduling for EDGE architectures," International Conference on Parallel Architecture and Compiler Techniques, 2004, PACT–13, pp. 74–84.

[Nag11] T. Nagatsuka, Y. Sakaguchi, T. Matsumura, and K. Kise, "CoreSymphony: an efficient reconfigurable multi–core architecture," ACM SIGARCH Computer Architecture News, vol. 39, iss. 4, 2011, pp. 32–37.

[Nee00] H. Neefs, H. Vandierendonck, and K. De Bosschere, "A technique for high bandwidth and deterministic low latency load/store accesses to multiple cache banks," International Symposium on High–Performance Computer Architecture, 2000, HPCA–6, pp. 313–324.

[Ond99] S. Onder and R. Gupta, "Dynamic memory disambiguation in the presence of out-of-order store issuing," International Symposium on Microarchitecture, 1999, MICRO–32, pp. 170–176.

[Pag06] K. Pagiamtzis and A. Sheikholeslami, "Content–addressable memory (CAM) circuits and architectures: a tutorial and survey," IEEE Journal of Solid–State Circuits, vol. 41, iss. 3, 2006, pp. 712–727.

[Pal97] S. Palacharla, N. Jouppi, and J. Smith, "Complexity–Effective Superscalar Processors," International Symposium on Computer Architecture, 1997, ISCA–24, pp. 206–218.

[Pap96] D. B. Papworth and G. J. Hinton, "Method and Apparatus for State Recovery Following Branch Misprediction in an Out-of-Order Microprocessor," United States Patent no. 5586278, 1996.

[Par00] J.M. Parcerisa and A. Gonzalez, "Reducing wire delay penalty through value prediction," International Symposium on Microarchitecture, 2000, MICRO–33, pp. 317–326.

[Par03] I. Park, C.L. Ooi, and T. Vijaykumar, "Reducing design complexity of the load/store queue," International Symposium on Microarchitecture, 2003, MICRO–36, pp. 411–422.

[Par04] J.M. Parcerisa, "Design of Clustered Superscalar Microarchitectures," Polytechnic University of Catalonia, Department of Computer Architecture, Ph.D. Thesis, 2004.

[Pat85] Y.N. Patt, S.W. Melvin, W.W. Hwu, and M. Shebanow, "Critical Issues Regarding HPS, High Performance Microarchitecture," Workshop on Microprogramming, 1985, MICRO–18, pp. 109–116.

[Rac03] P. Racunas and Y.N. Patt, "Partitioned first–level cache design for clustered microarchitectures," International Conference on Supercomputing, 2003, ICS–17, pp. 22–31.

[Rei98] G. Reinman and B. Calder, "Predictive techniques for aggressive load speculation," International Symposium on Microarchitecture, 1998, MICRO–31, pp. 127–137.

[Rot97] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith, "Trace processors," International Symposium on Microarchitecture, 1997, MICRO–30, pp. 138–148.

[Rot05] A. Roth, "Store vulnerability window (SVW): re-execution filtering for enhanced load optimization," International Symposium on Computer Architecture, 2005, ISCA–32, pp. 458–468.

[Saz97] Y. Sazeides and J. E. Smith, "The Predictability of Data Values," International Symposium on Microarchitecture, 1997, MICRO–30, pp. 248–258.

[Sal05] P. Salverda and C. Zilles, "A criticality analysis of clustering in superscalar processors," International Symposium on Microarchitecture, 2005, MICRO–38, p. 12 pp.

[Sub06a] S. Subramaniam and G. H. Loh, "Fire–and–Forget: Load/Store Scheduling with No Store Queue at All," International Symposium on Microarchitecture, 2006, MICRO–39, pp. 273–284.

[Sub06b] S. Subramaniam and G. H. Loh, "Store vectors for scalable memory dependence prediction and scheduling," International Symposium on High–Performance Computer Architecture, 2006, HPCA–12, pp. 65–76.

[San03] K. Sankaralingam, S. Keckler, W. Mark, and D. Burger, "Universal mechanisms for data–parallel architectures," International Symposium on Microarchitecture, 2003, MICRO–36, pp. 303–314.

[San03] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, Jaehyuk Huh, D. Burger, S. Keckler, and C. Moore, "Exploiting ILP, TLP, and DLP with the polymorphous

TRIPS architecture," International Symposium on Computer Architecture, 2003, ISCA–30, pp. 422–433.

[Sat06] Y. Sato, K. Suzuki, and T. Nakamura, "Power and Performance Advantages of the Highly Clustered Microarchitecture," International Workshop on Advanced Low Power Systems, 2006.

[Set03] S. Sethumadhavan, R. Desikan, D. Burger, C. Moore, and S. Keckler, "Scalable hardware memory disambiguation for high ILP processors," International Symposium on Microarchitecture, 2003, MICRO–36, pp. 399–410.

[Set07] S. Sethumadhavan, F. Roesner, J.S. Emer, D. Burger, and S.W. Keckler, "Late–binding: enabling unordered load–store queues," International Symposium on Computer Architecture, 2007, ISCA–34, pp. 347–357.

[Sez99] A. Seznec and P. Michaud, "De-aliased hybrid branch predictors," Technical Report RR–3618, Inria, 1999.

[Sha05] T. Sha, M. Martin, and A. Roth, "Scalable store–load forwarding via store queue index prediction," International Symposium on Microarchitecture, 2005, MICRO–38, pp. 12.

[Sha06] T. Sha, M. Martin, and A. Roth, "NoSQ: Store–Load Communication without a Store Queue," International Symposium on Microarchitecture, 2006, MICRO–39, pp. 285–296.

[Sim95] M. Simone, A. Essen, A. Ike, A. Kishamoorthy, T. Maruyama, N. Patkar, M. Ramaswami, M. Shebanow, V. Thirumalaiswamy, and D. Tovey, "Implementation Trade-offs in Using a Restricted Data Flow Architecture in a High Performance RISC Microprocessor," International Symposium on Computer Architecture, 1995, ISCA–22, pp. 151–162.

[Ska03] K. Skadron, M. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan, "Temperature–aware microarchitecture," International Symposium on Computer Architecture, 2003, ISCA–30, pp. 2–13.

[Ska97] K. Skadron and D. Clark, "Design issues and tradeoffs for write buffers," International Symposium on High–Performance Computer Architecture, 1997, HPCA–3, pp. 144–155.

[Smi81] J. E. Smith, "A study of branch prediction strategies," International Symposium on Computer Architecture, 1981, ISCA–8, pages 135–148.

[Sri02] V. Srinivasan, D. Brooks, M. Gschwind, P. Bose, V. Zyuban, P. Strenski, and P. Emma, "Optimizing pipelines for power and performance," International Symposium on Microarchitecture, 2002, MICRO–35, pp. 333–344.

[Sta00] J. Stark, M. D. Brown, and Y.N. Patt, "On Pipelining Dynamic Instruction Scheduling Logic," International Symposium on Microarchitecture, 2000, MICRO–33, pp. 57–66.

[Sto05] S. Stone, K. Woley, and M. Frank, "Address–indexed memory disambiguation and store–to–load forwarding," International Symposium on Microarchitecture, 2005, MICRO–38, pp. 12.

[Ten01] J. M. Tendler, S. Dodson, S. Fields, Hung Le, B. Sinharoy, "POWER4 System Microarchitecture," International Business Machines, Technical White Paper, 2001.

[Tor05] E. Torres, P. Ibanez, V. Vinals, and J. Llaberia, "Store buffer design in first–level multibanked data caches," International Symposium on Computer Architecture, 2005, ISCA–32, pp. 469–480.

[Tse06] J.H. Tseng, "Banked Microarchitectures for Complexity–Effective Superscalar Microprocessors," Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, Ph.D. Thesis, 2006.

[Tys97] G. Tyson and T. Austin, "Improving the accuracy and performance of memory communication through renaming," International Symposium on Microarchitecture, 1997, MICRO–30, pp. 218–227.

[Vij04] T. Vijaykumar and Z. Chishti, "Wire delay is not a problem for SMT (in the near future)," International Symposium on Computer Architecture, 2004, ISCA–31, pp. 40–51.

[Viv07] R. Vivekanandharn and R. Govindarajan, "A Scalable Low Power Store Queue for Large Instruction Window Processors," International Conference on Parallel Architecture and Compilation Techniques, 2007, PACT–16, p. 430.

[Wat05] S. Watanabe, H. Irie, M. Takada, and S. Skai, "Reducing Issue Delay of Store Instructions on A clustered Microarchitecture," Information Processing Society of Japan SIG Technical Reports, vol. 2005, no. 19, 2005–ARC–162, 2005–HPC–101, 2005, pp. 199–204.

[Wat10] Y. Watanabe, J. D. Davis, D. A. Wood, “WiDGET: Wisconsin Decoupled Grid Execution Tiles,” International Symposium on Computer Architecture, 2010, ISCA–37, pp. 2–13.

[Wea94] D. L. Weaver,T. Germond, “The SPARC Architecture Manual, Version 9”, SPARC International, 1994.

[Web02] D. A. Webb, J.B. Keller, and D.R. Meyer, “Data cache having store queue bypass for out–of–order instruction execution and method for same,” United States Patent no. 6360314, 2002.

[Wen07] T.F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, “Mechanisms for store–wait–free multiprocessors,” International Symposium on Computer Architecture, 2007, ISCA–34, pp. 266–277.

[Woo08] Dong Hyuk Woo and Hsien-Hsin S. Lee, "Extending Amdahl’s Law for Energy–Efficient Computing in the Many–Core Era," IEEE Computer December 2008, pp. 24–31.

[Yeh92] T. Y. Yeh and Y. N. Patt. “Alternative implementations of two–level adaptive branch prediction,” International Symposium on Computer Architecture, 1992, ISCA–19, pp 124–134.

[Yin05] Yingmin Li, K. Skadron, D. Brooks, and Zhigang Hu, “Performance, energy, and thermal considerations for SMT and CMP architectures,” International Symposium on High Performance Computer Architecture, 2005, HPCA–11, pp. 71–82.

[Yoa99] A. Yoaz, M. Erez, R. Ronen, and S. Jourdan, “Speculation techniques for improving load related instruction scheduling,” International Symposium on Computer Architecture, 1999, ISCA–26, pp. 42–53.

[You95] C. Young, N. Gloy, and M.D. Smith, “A comparative analysis of schemes for correlated branch prediction.” International Symposium on Computer Architecture, 1995, ISCA–22, pp. 276–286.

[Zha05] M. Zhang and K. Asanovic, “Victim replication: maximizing capacity while hiding wire delay in tiled chip multiprocessors,” International Symposium on Computer Architecture, 2005, ISCA–32, pp. 336–345.

[Zyu00] V.V. Zyuban, “Inherently Lower–Power High–Performance Superscalar Architectures,” Ph.D. dissertation, CSE Dept., Univ. of Notre Dame, 2000.

[Zyu01] V.V. Zyuban and P.M. Kogge, “Inherently Lower–Power High–Performance Superscalar Architectures,” International Symposium on High Performance Computer Architecture, 2001, HPCA–10, pp. 268–285.

LIST OF FIGURES

CHAPTER 1. INTRODUCTION

CHAPTER 2. PREVIOUS RELATED WORK

CHAPTER 3. BANK PREDICTORS

CHAPTER 4. A DISTRIBUTED MEMORY UNIT

CHAPTER 5. IMPROVEMENTS TO THE DISTRIBUTED MEMORY UNIT

CHAPTER 6. CONCLUSIONS

LIST OF TABLES

CHAPTER 1. INTRODUCTION

CHAPTER 2. PREVIOUS RELATED WORK

CHAPTER 3. BANK PREDICTORS

CHAPTER 4. A DISTRIBUTED MEMORY UNIT

CHAPTER 5. IMPROVEMENTS TO THE DISTRIBUTED MEMORY UNIT

CHAPTER 6. CONCLUSIONS

LIST OF EXAMPLES

CHAPTER 1. INTRODUCTION

CHAPTER 2. PREVIOUS RELATED WORK

CHAPTER 3. BANK PREDICTORS

CHAPTER 4. A DISTRIBUTED MEMORY UNIT

CHAPTER 5. IMPROVEMENTS TO THE DISTRIBUTED MEMORY UNIT

CHAPTER 6. CONCLUSIONS